International Graduate School
Dynamic Intelligent Systems

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Full Semantics Preservation in Model Transformation

by

Maria Semenyak

maria.semenyak@upb.de

PhD Thesis

in partial fulfilment of the requirements for the degree of
doctor rerum naturalium (Dr. rer. nat.)
supervised by

Prof. Dr. Gregor Engels

co-supervised by

Prof. Dr. Heike Wehrheim
Prof. Dr. Leena Suhl

Paderborn, August 24, 2011

Abstract

Model transformations play a key role in automated software development processes, where the modern trend is directed towards specification of software with an abstract model and its step-wise transformation into code. It is important that the code meets the initial specification. Therefore, the question for a model transformation is whether the transformed model fulfills the behavioural properties of the initial model.

In this thesis, a method is presented for proving that a specified model transformation is semantically correct in the sense that it preserves all behavioural properties of a source model. The method is introduced within the model-driven architecture approach, where syntax of a modelling language is defined by a meta-model. A meta-model is specified by means of object-oriented approaches, which based on the diagrams that are basically graphs, sometimes attributed with textual information. This allows a meta-model to be considered as a graph. Then, the model transformation is defined with the triple graph grammar technique.

We deal with languages, the behavioural semantics of which can be formally specified by means of graph transformation systems too. Therefore, the graph transformations are used twice in this thesis: for model transformation and for behavioural semantics specification. An application of a graph transformation system to a graph that stands for a model results a Labelled Transition System (LTS) generation. An LTS represents a set of all possible transition sequences, which is supposed to describe behaviour of the system.

A temporal logic called ACTL (an Action-based version of CTL*), is used to specify the behavioural properties of a model on an LTS. Full preservation of behavioural properties means that an LTS generated for *any* source model and the LTS generated for the corresponding target model cannot be distinguished by the truth value of ACTL formulas. When the same ACTL properties hold for two LTSs, then the LTSs are called ACTL equivalent.

The method introduced in this thesis is based on establishing an equivalence relation over LTSs called weak bisimulation, which is a standard concept from concurrency theory, normally used to compare executions and to decide whether they are indeed equivalent or not. In this thesis, it is shown that weak bisimulation on LTSs implies ACTL equivalence. Then, showing the behaviour preservation during model transformation requires establishing a weak bisimulation on the LTSs of source and target models.

As a summary, the method provides the guidelines how to specify the modelling languages and model transformation in order it could be possible to show that the specified model transformation is semantically correct. The method also explains how to define an equivalence relation over LTSs and to carry out the proof that the

defined equivalence relation is a weak bisimulation. Additionally, we explain how the behavioural properties of a source model can be interpreted for a target language by usage of the triple graph grammar transformation and the weak bisimulation.

In addition, the method is validated by a case study, where we define a model transformation between two modelling languages: Calculus of Communication Systems (CCS) and Petri nets. The main goal of the case study is to show how to proof that the defined model transformation is semantics preserving and to illustrate the specification of sample behavioural properties for the CCS and their interpretation for the Petri nets language.

Zusammenfassung

Modelltransformationen spielen eine Schlüsselrolle in automatisierten Software-Entwicklungsprozessen. Der moderne Trend geht in Richtung Spezifikation der Software mit einem abstrakten Modell und dessen schrittweiser Verwandlung zum Programmiercode. Es ist sehr wichtig, dass der Programmiercode der ursprünglichen Spezifikation entspricht. Deswegen ist eine wichtige Frage einer solchen Modelltransformation, ob das transformierte Modell die Verhaltenseigenschaften des Ausgangsmodells erfüllt.

In dieser Doktorarbeit wird eine neue Methode für die Spezifikation von einer Modelltransformation entwickelt und es wird gezeigt, dass die zu Grunde gelegte Modelltransformation korrekt ist. Dabei bedeutet Korrektheit, dass die Verhaltenseigenschaften bei der Modelltransformation erhalten bleiben. Die Methode wird in dem Ansatz der modellgetriebenen Architektur eingeführt, wo ein Metamodell für modellierende Sprachen definiert wird. Ein Metamodell ist spezifiziert anhand objektorientierter Ansätze, wo die Diagramme auf Graphen basieren und sind manchmal mit textuellen Informationen angereichert. Von daher kann ein Metamodell auch als ein Graph bezeichnet werden. Die Modelltransformation ist dann anhand von Triple Graph Grammatik definiert.

In dieser Doktorarbeit werden Sprachen benutzt, die Verhaltenssemantik beinhalten, welche wiederum mit der Graph Transformation System Methode definiert sein kann. Hierin wird die Graph Transformation zweimal verwendet: Spezifikationen für Modelltransformation und für Verhaltenssemantik. Als Ergebnis der Anwendung der Graph Transformationen zu dem Graph, der ein Modell bezeichnet, wird ein Labelled Transition System (LTS) erzeugt. Ein LTS vertritt einen Satz von allen möglichen Transitionsfolgen, die das Systemverhalten beschreiben sollen.

Eine temporale Logik, die ACTL (Aktionbasierte Logik von CTL*) genannt wird, wird verwendet, um die Verhaltenseigenschaften des Modells in ein LTS zu spezifizieren. Vollständige Erhaltung der Verhaltenseigenschaften bedeutet, dass ein LTS, welches für *jede* Instanz von Modell erzeugt wurde, und weiteres LTS, generiert für ein korrespondierendes Zielmodell, können nicht unterschieden werden durch den Wahrheitswert von ACTL Formeln. Wenn beide LTS die gleichen ACTL Eigenschaften aufweisen, dann werden diese LTSs ACTL äquivalent genannt.

Die Methode basiert auf dem Ansatz einer Äquivalenzbeziehung (schwache Bisimulation) über LTS, welche das Standardkonzept in der concurrency Theorie ist. Normalerweise benutzt man diese Äquivalenzbeziehung um Berechnungen zu vergleichen und zu entscheiden, ob sie tatsächlich äquivalent sind oder nicht. In dieser Doktorarbeit wird gezeigt, dass schwache Bisimulation auf LTS die ACTL Äquivalenz impliziert. Um die Erhaltung der Verhaltenseigenschaften während der Modelltransformationen zu reifen, ist es erforderlich der schwache Bisimulation auf

LTS von Anfangs- und Ausgangsmodellen zu erstellen.

Zusammenfassend bietet die Methode eine Leitlinie, wie Modellierungssprachen und Modelltransformationen spezifiziert werden können, damit es möglich ist zu zeigen, dass spezifizierte Modelltransformation semantisch korrekt ist. Sie führt einen Beweis aus der zeigt, dass die definierte Äquivalenz eine schwache Bisimulation ist. Zusätzlich wird erklärt wie die Verhaltenseigenschaften eines Ausgangsmodell für eine Zielsprache interpretiert werden können durch die Benutzung von Triple Graph Grammatik Transformation und der schwache Bisimulation.

Ferner wurde die Methode mit einer Fallstudie bestätigt, wo eine Modelltransformation zwischen zwei Modellsprachen definiert wurde: Calculus of Communication Systems (CCS) und Petri Netze. Das Hauptziel der Fallstudie ist, zu zeigen, wie man beweist, dass die definierte Modelltransformationen das Semantikverhalten beibehält. Sie illustriert die Spezifikation der Verhaltenseigenschaften für CCS und die Interpretation bezüglich der Petri-Netz-Sprache.

Acknowledgements

My way to this PhD was very long and excited. It was accompanied with my move to Germany, very unfamiliar and full of surprises country with mystery people. In that early ages I was afraid of nothing and was open to all forthcoming adventures and challenges. Luckily, there were many of them. Some of them made me quite sad sometimes, but now I remember all gathered experience with a smile. If somebody will propose to repeat these long path, I would certainly agree!

Among the numerous people, which accompanied me during last years, firstly I would like to thank Prof. Dr. Gregor Engels for choosing me among the other candidates and thereof giving me a chance to work in his working group. Being a very busy professor, Prof. Dr. Gregor Engels still managed to find a time to supervise me. Another important person is Prof. Dr. Heike Wehrheim, who invested time in working with me. Furthermore, I would like to thank Prof. Dr. Leena Suhl Prof. Dr. Kleine Büning, Dr. Theodor Lettmann for joining the board of examiners.

I am very grateful to the International Graduate School (IGS) for financial support of my studies. I am especially thankful to Prof. Dr. Eckhard Steffen (Director of IGS), who came in Novosibirsk to tell students about this program personally. My special thank is for the IGS team for the organization of different events, which gave me a chance to know the other PhD students and the area where they are specialized closer.

In the first two years of my studies I was involved in a project, where I worked with very interesting people, such as Prof. Dr. Heike Wehrheim, Prof. Arend Rensink, Christian Soltenborn, Prof Dr. Barbara König and Mathias Hülsbusch. I thank all of them for interesting scientific ideas, for introducing me their style of work and communication. During this time I have got a lot of experience working on the research topic.

For sure, I would like to mention my colleagues, both former and present. Besides providing critical views on my work, they introduced me a German culture and German traditions. In particular, I would like to thank Dr. Martin Assman, Dr. Alexander Förster, Christian Soltenborn, Dr. Joel Greenyer and my Pakistani colleague Zille Huma for a strong support, when I arrived in Germany. A very special thank is for our technician, Friedhelm Wegener, and secretary, Beatrix Wiechers, which make my stay in Germany pleasant. In this part it is appropriate to thank my assistants Erik Bonner and Wilfried Bröckelmann for investing their time for correcting my English and German respectively.

During my stay in Paderborn I became close to many people, who turn to be very reliable friends. They kept me confident (which is really important in my case). Special thanks to Mariana Reyes, Natalia Akchurina and Valentina Avrutova. Although I visited Russia only few times during my studies in Germany, I have still very good friends there, which stay close after passing so many years. Last years they supported me in a distance and, for some reasons, always believe in me, among them Elena (Un) Kim, Irina (Golovina) Molodih and Irina Posrednikova.

Finally, this work would not have been possible without unlimited support and encouragment from my partner Erik, who came unexpectedly in my life and since then makes me happy every day. A special thank to my family, for being always somewhere nearby.

Short contents

Contents

CHAPTER 1

Motivation and Overview

Software development is one of the serious growth areas in the IT industry. The important aspect of this area is to find a concept which effectively delivers the broad range of customer requirements and industrial demands into a qualitative solution. The rapid progress of the IT industry led to the point where technologies became very complex. Satisfying customer demands and delivering qualitative solution have turned a software development into a difficult process.

A qualitative solution is defined very broadly, but the main characteristics are that it must correspond to the modern standards, and be cheap and reliable. To reach these characteristics analysis of customer demands prior to implementation is required. Analysis helps to create a better view of a system before it is implemented. Analysis on the earlier stages of software development process assumes the specification of a system with a formal model and its *verification*. It has been shown [McC04] that a bug found in the early stages (such as requirements specification or design) is cheaper in terms of money, effort and time, to fix than the same bug found later on in the process.

The goal of the verification process within the software development process is formally proving the correctness of a design model with respect to a certain formal property. There are different verification methods. In one such mehtod a design model has an underlying formal model and the software properties are specified with a formal language, e.g. with a Computation Tree Logic [BCG88], according to customer requirements. Then the properties are verified against the formal model.

Before we go deep into detail, we provide a small case-study of a software development process based on the famous V-Model development method [VMO] (see Figure 1.1). This method proposes a sequential process for system design and verification and consists of several *stages*. The process starts with a definition of concrete

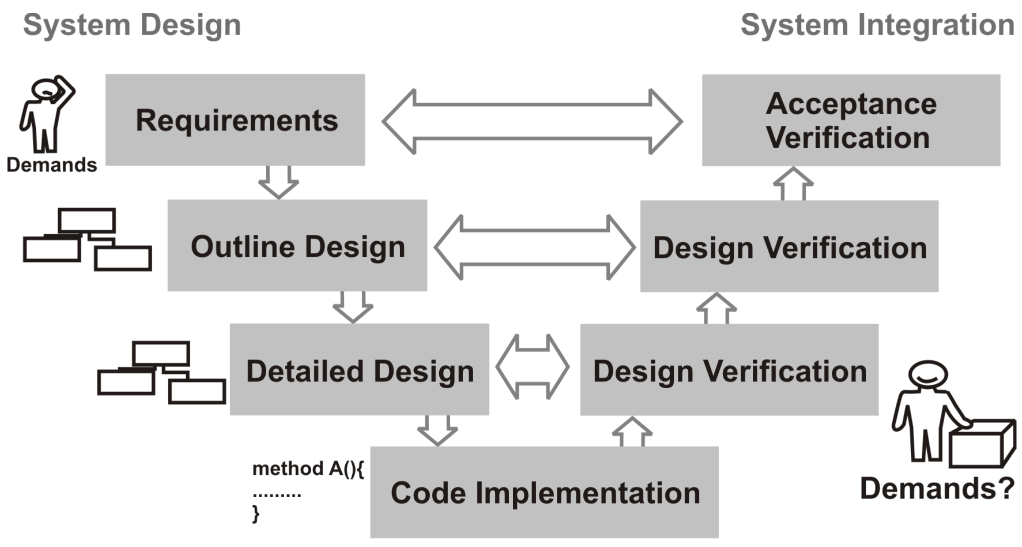


Figure 1.1: The V-Model development method

requirements derived from customer demands. The next stage is an *outline design model* of the system on an abstract level, which is followed by a *detailed design model*. On each step the design is *verified*. The reason for this is that verification of the design model is mostly cheaper than verification and validation of an implemented system. The *implementation* stage in which code is generated is not the last, because there is a need to check if the system works as the customer required. Therefore, the software is verified against primary design. The final step is an *acceptance verification*, where the customers demands are *verified*. In case the customer is not satisfied, the process of system design is repeated.

In multi-stage software development process a design model is not always specified within one modelling language. In order to ensure that a model on the next stage fulfills the same properties as its predecessor, the properties must be *interpreted* and then verified (see Figure 1.2).

Among the variety of requirements specified for the system, there are some that describe the behaviour of the system. For example, “the program must always terminate” is a requirement concerning the behaviour. Specified properties that describe such requirements are called *behavioural properties*. To *verify* the behavioural property we need formal *behavioural semantics*, which describe the meaning of a design model. In this thesis we use executable behavioural semantics. That means we can get a formal model of behaviour which displays the transitions. Then behavioural properties could be verified towards a formal behavioural model.

Verification of models brings a lot of advantages to the software development process, however the progress has not held yet. The modern tendency is towards automation of the software development process, where an automated model trans-

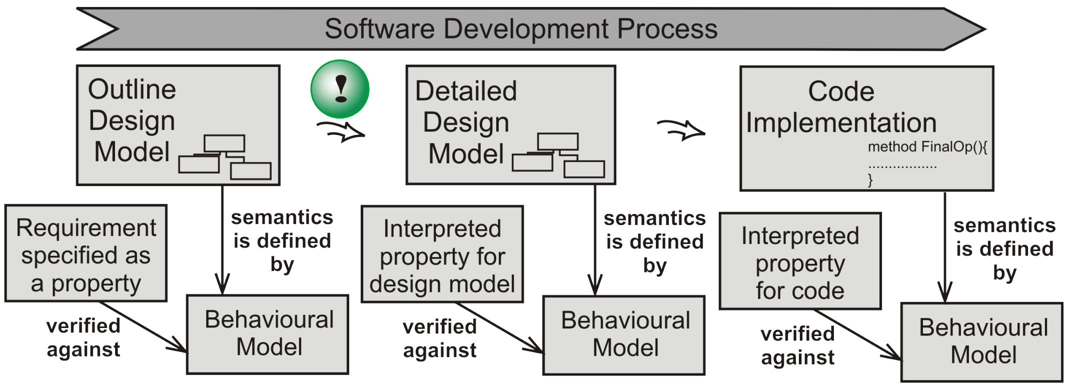


Figure 1.2: Idea of model transformation verification (green circle with an exclamation mark)

formation plays an important role. The task of partial or full model transformation is a conversion of a source model into a target model (see Figure 1.2 the exclamation mark). This helps to reduce the costs and make the software development process more reliable. The next step on the way to full automation is verification of model transformation instead of verification of a target model.

This thesis addresses the idea of specification of model transformation between two stages of the software development process and ensuring its *correctness*, in the sense that behavioural properties are preserved during model transformation. The advantage is that firstly, there could be a lot of work done automatically, and secondly, there is no need to verify a design model on the next stage.

1.1 Role of Models in Software Development

Software development tends to be complex not only in the sense of technology complexity, but also because of its interdisciplinary nature, i.e stakeholders with different backgrounds are involved. In order to reach a common understanding about a problem, there is a need to describe a problem using terms that are familiar to people who work in the domain of the problem, rather then in terms only familiar to IT experts.

Among existing concepts of information representation, models are widely used for elaborate tasks. Models are attractive because they allow a problem to be precisely described in a way that avoids going into technological detail. Thus, models are normally more willingly accepted by domain experts than programming languages for example. Being defined as simple constructions, mostly with graphical interface, models can be written in formal notation and be involved in formal proofs. There are existing common standards that are used by domain experts and developers.

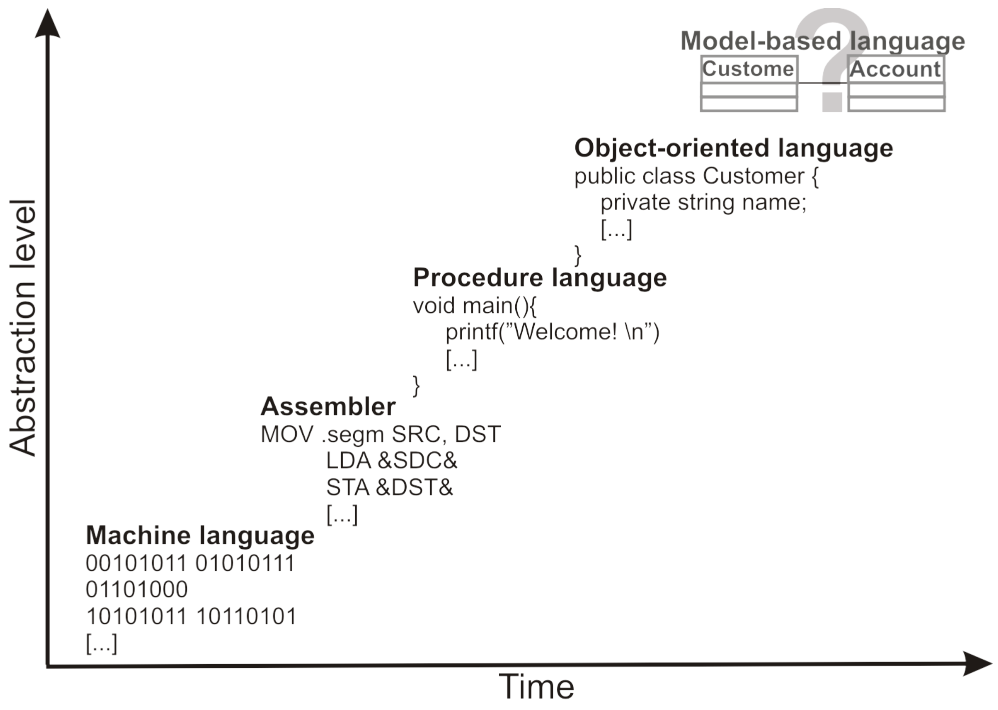


Figure 1.3: Incremental abstraction of programming languages (reproduced from [GPR06])

Models are brought to the software development process because they are a promising tool for overcoming the current problems of the software development process. One such problem is the dominance of technical issues. Since a variety of technologies are involved in software development, an overview of the main processes is hard to get without a simplified view. Another important issue concerns the necessity having different views of the system, such as internal and external parts of the projects.

For sure, models are not the only tool that should be used to overcome the problems addressed above, it is still possible to use the concepts based on textual descriptions. However, the general direction in which development of software languages is changing indicates a tendency towards increasing programming languages abstraction [GPR06]. In the past, the abstraction level from machine code was raised to higher level languages and to object oriented languages (Figure 1.3). According to the current trends, model-based languages are likely to be the next level of abstraction [Béz05].

Today, models are involved in the integral part of software development process design and its analysis. A growing collection of methods, techniques and theory for modelling help to find the solution for complicated tasks. In this thesis we show that

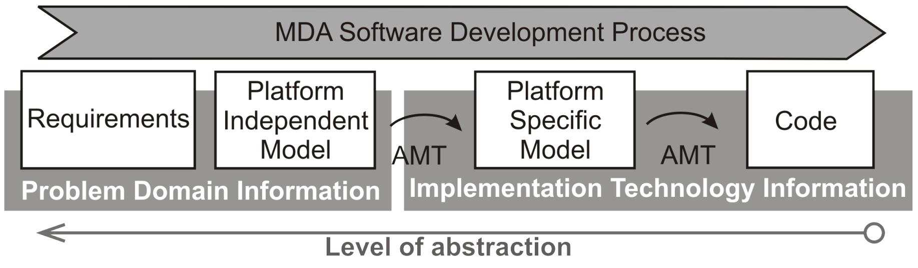

MDA Software Development Process
Requirements
Platform Independent Model
AMT
Platform Specific Model
AMT
Code
Problem Domain Information
Implementation Technology Information
Level of abstraction

Figure 1.4: Automated Model Transformation (AMT) in the MDA approach

models are a formal tool for software development design, which can be correctly turned into a less abstract language.

1.2 Model-Based Software Development Process

Automation plays an increasingly important role in the world economy. Engineers try to combine automated devices with mathematical tools to create complex systems for a rapidly expanding range of applications and human activities. An automated development process of a software development system entails many advantages, such as reduction of human interaction in technical issues, a cut in expenditure, time saving and concentration on the tasks requiring subjective understanding high-level tasks such as exploring problem domains.

Among the existing proposals for automated software development process solutions, the Model-Driven Architecture (MDA) [MDA, KWB03] approach is one of the favorite. The MDA approach is a model-based approach, which provides a set of guidelines for the structuring of specifications, which are expressed as models. The benefit of MDA is that it separates application knowledge from implementation technology by performing two different models for platform independent and platform dependent layers. Therefore, the potential for porting of a model to different platforms is high (see Figure 1.4).

The main idea of the MDA is to perform full or partial automated transformation of specification into code. This means that the designer must only provide a specification of a Platform Independent Model (PIM) on the abstract level, then a Platform Specific Model (PSM) and executable code can be automatically generated. In context of the MDA, model transformation is a set of rules that together describe how source models are transformed into target models, where a target model could also be code.

There are two types of model transformation in the MDA: *horizontal* and *vertical model transformations* [Chr04, RPH^{+}03]. The former are used to restructure a model in order to transform a model to a more technical level, therefore affecting two levels of abstraction. The latter, by contrast, keeps the models on the same level of abstraction. They are used to refine or to abstract a model during forward or reverse engineering.

The main problem of the automated MDA approach today is a big semantic gap between the business requirements and execution code. The division of the software development process into task units according MDA concepts still leaves a big gap between PIM and PIM, or PIM and PSM. Unlike a transformation, e.g., a transformation from PSM to code (since PSM is very close to code), inter-model transformation are not easy to realize. In the following, simple example demonstrates this problem.

Model transformation definition assumes the existence of a *mapping* between elements of transformed models. On the intuitive level mapping is defined for the elements that perform similar behaviour. The thing is that even if the mapping is set for some elements, the correctness of model transformation is not guaranteed. For example, one could assume that the `Token` element of the UML Activity diagrams [UMLb] could be mapped to the program counter if the model transformation between the UML Activities and programming language is needed. The assumption will be only partially correct, because UML Activity specification allows existence of several `Token` elements during run-time, however the existence of two similar program counters would have another meaning.

The problem of semantic gap between modelling languages could be solved to a considerable extent by specifying model transformation in a special way. The specification of model transformation must be possible to verify, which means to ensure that it preserves a class of properties.

In this thesis we propose a solution for specification of model transformation and ensuring that the specified model transformation is correct. Thus, we emphasize two stages in the software development process (Figure 1.5). These stages are performed by two actors. Firstly, a model transformation expert, who is well informed about PIM and PSM specification languages, specifies a model transformation. Secondly, a quality assurance expert in verification technologies performs a formal proof of its correctness.

1.3 Correctness of Model Transformation

A model transformation is called *correct* when the transformed model still fulfills the properties of the source model (see Figure 1.6). For example, if a source model guarantees that a program must terminate, then the target model must always guarantee this property too.

Formal verification methods help to ensure the correctness of model transformation. The necessity of formal verification methods is growing each day due to the size of software systems and complexity of error detection. The primary ideas of these methods is to prevent deviation of software from its expected behavior.

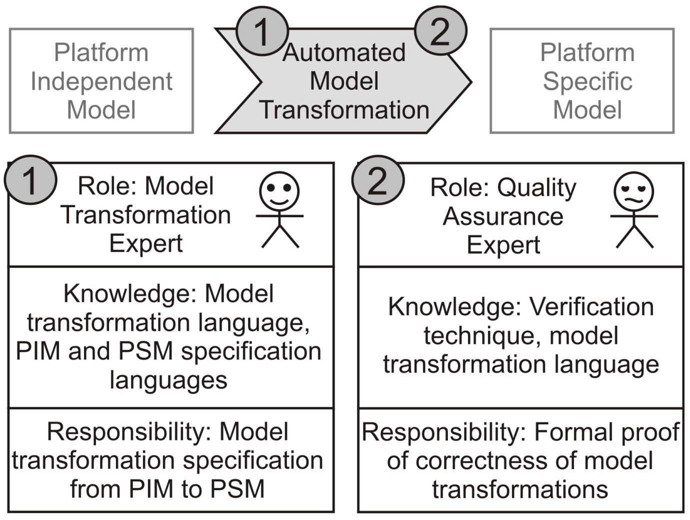

Platform Independent Model
1
Automated Model Transformation
2
Platform Specific Model
1
Role: Model Transformation Expert
Knowledge: Model transformation language, PIM and PSM specification languages
Responsibility: Model transformation specification from PIM to PSM
2
Role: Quality Assurance Expert
Knowledge: Verification technique, model transformation language
Responsibility: Formal proof of correctness of model transformations

Figure 1.5: Automated model transformation and its verification

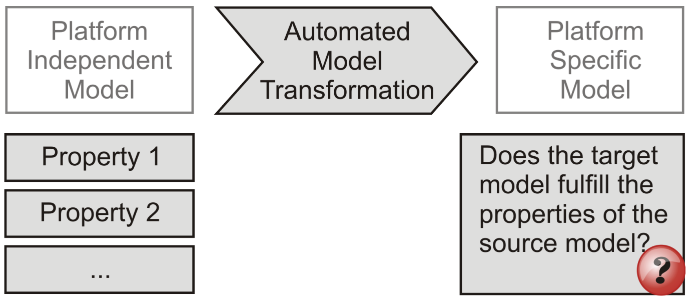

Platform Independent Model
Automated Model Transformation
Platform Specific Model
Property 1
Property 2
...
Does the target model fulfill the properties of the source model?
?

Figure 1.6: The main question for an automated model transformation

Although, the MDA approach concept helps to overcome many problems related to the software development process, the MDA does not give a precise answer on how to ensure that a model transformation is correct regarding the preservation of behavioral properties. This fact motivates people to find a solution on their own.

So far, there has not been a great deal of investigating into verification techniques that could give a guarantee about preservation of behaviour properties, in particular in those cases where the source and target languages are different. There exist several approaches in the area of compiler [Gle03, NL97], where a compiler has a role of model transformation and the correctness of going from code to a lower level code is ensured. Since model to code transformations are different to inter-model transformations due to the different nature of compiler and modelling languages, the techniques used in compilers are not directly applicable in our case.

This thesis provides a solution for specification of model transformation between modelling languages and ensuring its correctness in the sense of behavioural properties preservation. We consider a strong criteria of correctness: preservation of *all* behavioural properties for *all* models defined by some language.

1.4 Objective of this Thesis

The objective of this thesis is to supply a method for the specification of model transformation and ensuring that the specified model transformation is correct in the sense that it preserves behavioural properties. The method must fit the standards of modern modelling languages equipped with formal behavioural semantics. It must give a guarantee of correctness for a preservation of a broad class of behavioural properties. The method must be well adopted within some community.

The method must be in particular illustrated on a concrete example of existing languages: Calculus of Communication Systems (CCS) [Mil95] and Petri nets [Rei85]. The languages fit the idea of horizontal model transformation in the MDA approach. Both languages are used for model specification on a platform independent level. The CCS language is used for modelling concurrency, and Petri nets are widely used to simulate nondeterministic computation. The main requirement for the case study is to apply our method and additionally illustrate the preservation of behavioural properties for the target model.

1.5 Solution Idea

In this thesis we propose a solution for ensuring behaviour preservation during model transformation. The solution consists of the following steps:

(I) Formal specification of source and target models,
(II) Formal specification of model transformation,
(III) Comparison of behavioural models and proving that the behavioural properties of a source model are preserved during model transformation.

For formal specifications (I) and (II), we decided to use graph transformation rules [BH02, Roz97], because the graph transformation approach has a number of advantages. First of all, graph transformation rules are specified completely formally; this is important for our final goal of proving that our transformation is behaviour preserving. Second, due to their visual appearance, graph transformation rules are relatively easy to understand. Moreover, due to the availability of tools for graph transformation rules, the transformation is executable.

We specify syntax of the modelling languages by means of graphs and the behavioural semantics by means of graph transformation rules. In this case, graph transformation rules describe the changes of a model. The specification of syntax with graph allows to specify model transformation with graph transformation rules.

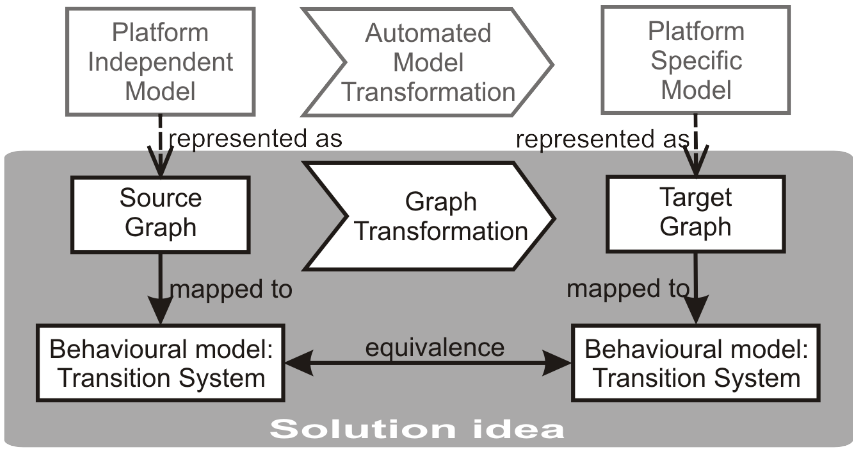


Figure 1.7: General idea for solution

Behavioural semantics specified by means of graph transformation rules generate a *transition system* by step-wise application to a source graph. A transition system then serves as a domain for comparison of the behavioural semantics, i.e. a transition system is a basis for step (III).

A transition system is a perfect common domain for comparison of different modelling languages with formal behavioural semantics (see Figure 2.15). This is because after a transition system is generated, both models fulfill (or do not fulfill) the same observable properties at the same abstraction level. Establishing an equivalence relation over the transition systems leads to preservation of some class of properties.

In this thesis we are going to study branching-time behaviour over transition system. Weak bisimulation [BK08] is one such equivalence relation. It aims to identify transition systems with the equivalent branching structures, and which can simulate each other in a stepwise manner. Roughly speaking, a transition system TS_1 can simulate transition system TS_2, if every step of TS_1 can be matched by one (or more) steps in TS_2. Weak bisimulation denotes the possibility of mutual stepwise simulation. Since weak bisimulation is one of the most discriminating notions of behavioural equivalence (essentially preserving all properties in any reasonable temporal logic), we call this *full* semantic preservation.

In this thesis we also explain how behavioural properties can be specified over the transition system. For this, we use a temporal logic, where graph transformation rules play a role of atomic propositions. Then we use the weak bisimulation to interpret the properties of a source model for a target model.

1.6 Structure of this Thesis

The core of this thesis is the introduction of a method for proving the correctness of model transformation, which fulfills the above stated requirements. The presentation includes an explanation of the involved techniques.

In Chapter 2, we provide an in-depth introduction to the MDA architecture with a definition of the problem we try to solve. We specify formal problem statement and give a reader an idea how we are going to solve it.

Chapter 3 explains the basic notions of graph transformation. Additionally, there is a discussion of application area of graph transformation, such as definition of abstract syntax for modelling languages, as well area for description of behavioural semantics and model transformation by means of graph transformation.

In Chapter 4 we elaborate the ideas of behavioural properties verification. The goal of this chapter is to establish a connection between (1) the class of properties, which hold for some transition system, and (2) an equivalence relation over transition systems. By showing that (2) implies (1), we achieve the objective of our method.

Our method for specification of model transformation and ensuring its correctness is explained in Chapter 5. The method is illustrated on two sample languages. Additionally, it is shown how some properties of a source model could be interpreted for a target model.

The applicability of the proposed method is demonstrated in Chapter 6, which contains elaborate case study targeting model transformation between two real languages: CCS and Petri nets.

Chapter 7 concludes this thesis. It provides a summary of the presented approach, gives a critical look at the proposed method and draws perspectives for future work.

CHAPTER 2

Problem Statement

The focus of this thesis is the behaviour preserving model transformation within the Model-Driven Architecture (MDA) [MDA, PM07] approach, which is a promising model-based approach for application design and implementation of software systems. This chapter provides an in-depth introduction to this topic, elaborating on statements made in Chapter 1.

We start with a description of the MDA approach, in which we indicate a problem connected with reliability of model transformation (Section 2.1). To increase the reliability we propose to ensure the preservation of behavioural properties during model transformation. We specify the requirements for behaviour preserving model transformation in Section 2.2. Based on these requirements, we present a survey of existing approaches in Section 2.3. Because non of the existing approaches fulfill all our requirements, we investigate a method for ensuring the behavioural semantics correctness of model transformation (Section 2.4).

2.1 Model-Driven Architecture Approach

The MDA standard is suggested by the Object Management Group (OMG)[1]. It encourages efficient use of models in the software development process and is becoming widely referenced in the industry.

The MDA specification provides a solution for a wide range of problems connected with the software development process. The main difference from the tra-

[1]OMG specification is based on an ANSI/IEEE standard 1471 [GPR06], the shorten form of ‘Recommended Practice for Architecture Description of Software-Intensive Systems‘. In 2007 this standard was adopted by ISO/IEC JTC1/SC7 as ISO/IEC 42010:2007, ‘Systems and Software Engineering - Recommended practice for architectural description of software-intensive systems‘.

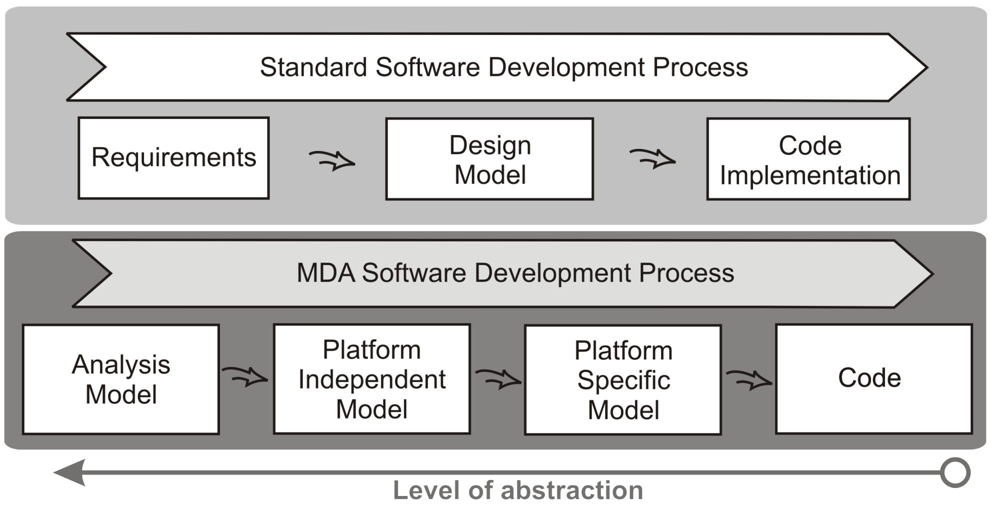


Figure 2.1: A standard approach for software development vs. the MDA approach

ditional software development cycle is that the MDA approach supports software development on a model-based basis, where each stage is a different level of abstraction. These levels fit into two categories: platform independent and platform specific. The OMG defines a Platform Specific Model (PSM) as a model that either has a platform neutral character (e.g., object oriented programming, component based development) [GPR06], is implemented within the concepts that involve a specific technology of a certain standard (e.g., Common Object Request Broker Architecture, CORBA, Java Enterprise Edition, Java EE), or is implemented for technology with specific concepts of one particular manufacturer (e.g., Microsoft.NET, IBM WebSphere, Java EE). The idea of MDA is to separate a platform independent level from platform specific details and thus remain as close as possible to the problem domain.

A Platform Independent Model (PIM) is specified without platform specific details, thereby remaining as close as possible to the problem domain. On different stages of abstraction (see Figure 2.1) PIM is extended with platform specific details and finally turning into PSM. Thus, by using MDA, it is expected that the final solution will closely address the original problem domain.

By introducing PIM and PSM to the software development process, the MDA approach provides two main advantages. Firstly, the MDA simplifies migration of applications to new platforms by systematic abstraction from technical aspects. The same PIM can be automatically transformed into multiple PSMs for different platforms (see Figure 2.2). Therefore, everything specified at the PIM level is completely portable. Secondly, shifting the focus from code to PIM directs attention to solving the business problem. This results in a system that better fits with the needs of the end user.

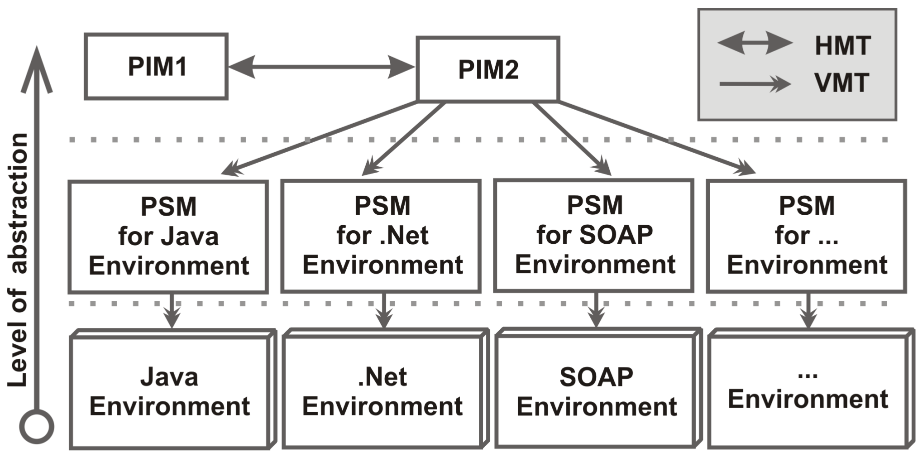


Figure 2.2: Vertical and Horizontal Model Transformations (VMT and HMT, respectively) in the MDA approach

The model-based software development suggested by the MDA can be described as follows. In the first stages of the MDA specification, the design model of a system is made on an abstract or a platform independent level. In the next stages, the PIM could be refined (PIM to PIM transformation in Figure 2.2), but finally the PIM evolves into a PSM. In the final stage, the code implementation is performed.

The main idea of the MDA approach is to automate the software development process. Therefore, model transformation plays an important role. A model transformation in the context of MDA is a formal model which defines how a source model could be transformed into a target model by means of rules. As can be seen in Figure 2.3, which depicts a fragment of the MDA meta-model for its main concepts, a model transformation defines the changes between PIM and PSM, PIM and PIM, PSM and PSM.

The concepts of stage-by-stage transformation described above for the software development process are not simple to implement. Unlike transformation from, e.g., PSM to code, the large semantic gap between modelling languages makes inter-model transformation a complex task. PIM and PSM normally are described within completely different domains and concepts. Even two PIM languages could differ a lot, which complicates the process of transformation. However, even when it is possible to define a model transformation, there is no standard method in the MDA specification to ensure the source model was transformed correctly. *Correctness* can be understood in many senses, such as syntactical correctness, static semantical correctness, behavioural correctness and others. The problem of the big semantic gap between modelling languages could be solved to a considerable extent, by showing preservation of some class of properties [EMHL03].

In this thesis, we specify a method for defining of a model transformation be-

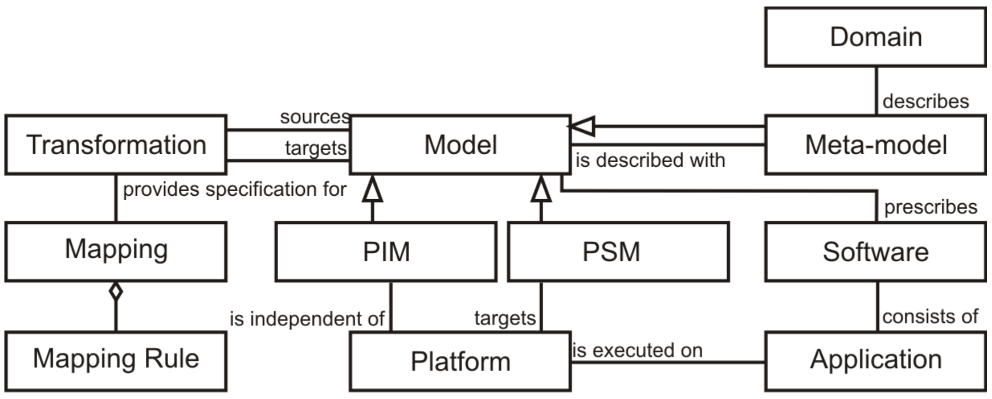


Figure 2.3: Fragment of the MDA central concepts

tween two different languages, and for ensuring the preservation of *behavioural* properties during model transformation.

The behavioural properties describe behaviour of a system and deal with notions such as *necessity, possibility* end *eventually.* An example of a behavioural property is a safety property, which is characterized as "nothing bad should happen". The mutual exclusion property [BK08] – always at most one process is in its critical section – is a typical safety property, or the traverse-to-completion property [Boc04], which characterizes the completion of one of the input pins. In approach [MCG04] such properties are also characterized as *mathematical properties*:

> *"If the transformation language or tool has a mathematical underpinning, it may be possible, under certain circumstances, to prove theoretical properties of the transformation such as termination, soundness, completeness, (syntactic and semantic) correctness, etc."*

With our thesis we extend the MDA approach with the concept of verification, i.e. showing the correctness of model transformations with respect to a formal specification using formal methods. In Figure 2.4 we add a *behavioural property* and a *software requirement* to the main concepts. A behavioural property represents a software requirement, which is described by *domain.* The behavioural property holds for a *model.* By verifying that behavioural properties hold for both source and target models we guarantee behavioural semantics preservation during model transformation.

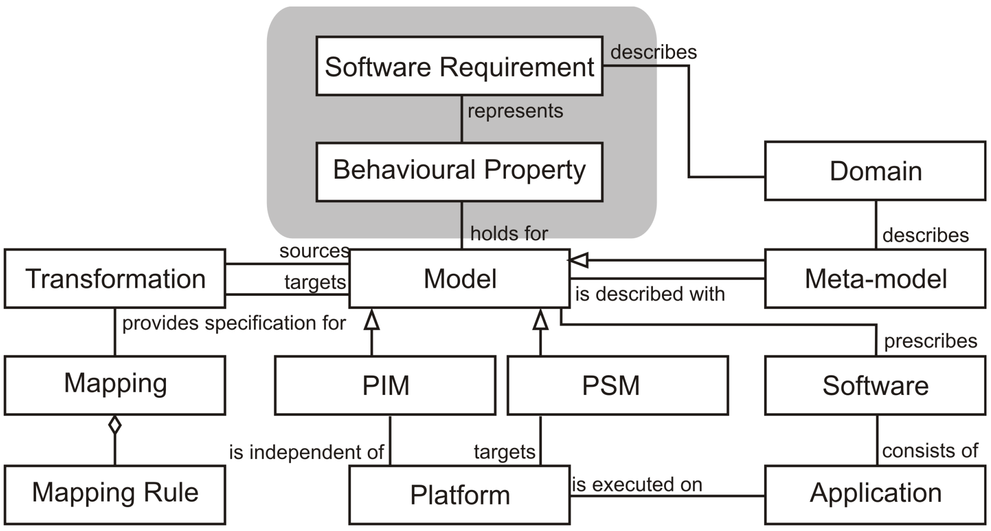


Figure 2.4: Extended MDA concepts

2.2 Requirements for Model Transformation

We consider the behavioural semantics preservation during model transformation to be a particularly important requirement. Formally, it means that we have two modelling languages: a source language L_1 and a target language L_2. Let M_1 be a source model of L_1 and M_2 be a target model of L_2 such that M_2 is a result of model transformation applied to M_1. We want that for a behavioural property, formalized as φ, and its interpretation χ in the language L_2, the following statement holds:

$$M_1 \models \varphi \quad \Rightarrow \quad M_2 \models \chi(\varphi) \tag{2.1}$$

here $M \models \varphi$ denotes that φ holds for M. In addition to this, we want to specify three additional requirements on φ.

1. The statement must hold for *any* property φ formalized within a language Σ.
2. The language Σ must be expressive enough to specify a wide range of behavioural properties.
3. The statement must hold for any model M_1 of L_1.

Note that the behavioural property φ is specified in the source language L_1. Therefore, it must be also explained the interpretation of the behavioural property φ for the target model, i.e. the meaning of function χ.

In terms of the given notation we provide a definition of a model transformation which fully preserves semantics (in the context of this thesis, we mean behavioural semantics).

Definition 1 (Full semantics preservation in model transformation)**.** Let MT be a model transformation, which transforms M_1 into M_2. We say further, that MT is a *model transformation which fully preserves semantics* iff Statement 2.1 holds for
(1) any model M_1 of the source language L_1, and
(2) every behavioural property φ that holds for M_1.

We also require that the model transformation satisfies the following properties:

Syntactical correctness The model transformation translates correct models of the source language into correct models of the target language (here correctness means that a model is consistet with its meta-model).

Definedness The model transformation is applicable to every model of the source language.

Uniqueness The model transformation defines a unique target model from a given source model.

Bidirectional transformations To transform the source model into a target model, and the inverse transformation to transform the target model into source model.

Understandability The technique for specification of model transformation should be intuitive and efficient to use, the same as the technique for the proof of its correctness.

Adequacy The technique for the definition of model transformation must be sufficiently well adopted within some community.

2.3 Survey of Techniques for Semantics Preserving Model Transformations

In this section we provide an overview of approaches that propose a solution for semantics preserving model transformation. We discuss the most important ones in detail before presenting a conclusion.

2.3.1 Overview of Specific Approaches

At present there are a number of solutions for model transformation [CH06, CH03]. We consider those that preserve behavioural semantics. A Basic Distinction (BD) can be made between approaches which
(a) aim for a general behaviour preserving model transformation technique (i.e. goaled to finding a universal solution),
(b) approaches from the area of compiler correctness,
(c) approaches for a special kind of model transformation called refactoring,
(d) approaches which pursue specific goals (e.g., verifying some particular behavioural properties, dealing with some particular models),

(e) approaches dedicated to testing of model transformations.

There are many papers, which pursue the (b), (c), (d), (e) goals, however there is little done among (a). Many approaches are focused on their respective goals and can not be expected to yield a universal solution. We give an overview comparison of approaches in this section.

We propose three characteristics for the classification of related work. The first characteristic, denoted as *purpose*, concerns the main distinctions (a)–(e) between approaches and specifies in general an idea of an approach.

Another important characteristic is a chosen *mechanism.* Mechanism should be interpreted here in a broad sense. It includes techniques, languages, methods, and so on. For example, the modern modelling language, the Unified Modelling Language (UML) [UMLb], could be expressed with different well-established formalisms. The graph transformation approach [Roz97, EEKR99, BCE$^+$99] is one of them. It includes formalism and a set of techniques applicable to model transformation [GGZ$^+$05, EEPT06b]. Thereby, the problem of correctness during model transformations could be better studied within some formal approach.

An *implementation* is a relevant characteristic for a model transformation. This characteristic aims the languages, to which the approach was applied. Mostly, it is a case study, which involves real languages and model transformation between them. The characteristic is important, since some specific features of languages, the complexity and other important things could be observed.

We provide a summary of concrete approaches in Table 2.1. For each approach we list its authors, a purpose, a mechanism and an implementation example. In the first column we also mention a letter, which is related to the main distinction criteria (a)-(e) described in the first paragraph of this section.

Table 2.1: Overview of approaches for a semantics preserving model transformation

BD	Authors	Purpose	Mechanism	Implementation
(a)	S.Glesner, J.Leitner [GGL$^+$06, Lei06]	The correctness of model transformation specification using a theorem prover	Triple graph grammar rules, Isabelle/HOL (High Order Logic)	Model transformation between Specification and Description Language (SDL) and Programmable Logic Controller (PLC)
(a)	B.König, M.Hülsbusch [HKH10, EK04, RKE07]	The correctness of model transformation specification by establishing a proof	Graph transformations with borrowed context	Model transformation of sample toy languages
(b)	S.Glesner [Gle03]	Program checking to ensure the correctness of compiler implementations	Program checking with certificates	Code generator based on graphs
(b)	G.C.Necula, S.P.Rahul [NR01]	Program checking with certificates	High-order logic programs, high-order logic interpreter	Java source programs compiled into Proof-Carrying Code (PCC)

Table 2.1: Overview of approaches for a semantics preserving model transformation

BD	Authors	Purpose	Mechanism	Implementation
(c)	G.Karsai, A. Narayanan [NK08, KN06]	Behavioural equivalence of the Statechart model and the EHA model with respect to reachability	Graph transformation rules implemented in GReAT[BNvBK06]	Model transformation of Statecharts into Extended Hierarchical Automata (EHA)
(c)	P.Barbosa, F.Ramalho, J.Figueiredo, A.Júnior, A.Costa, L.Gomes [BRF$^+$09]	Verification whether the model transformation preserves some properties like soundness	ATL: A model transformation tool [JABK08]	Model transformation of Petri net model into Petri net submodels
(c)	L.Baresi, K.Ehrig, R.Heckel [BEH06]	Verification of model transformation	Graph transformations, the AGG tool	A case study with BPEL
(c)	M.Proietti, A.Pettorossi [PP91]	To ensure that the transformed programs are equivalent to the initial ones, when evaluated by a Prolog interpreter or compiler	Logical theory	Prolog programs

Table 2.1: Overview of approaches for a semantics preserving model transformation

BD	Authors	Purpose	Mechanism	Implementation
(d)	Dániel Varró [Var02, Var04, EEL$^+$05]	Model transformation which are syntactically correct, the behavioural semantic properties could be proven by theorem provers and model checker	Graph transformation rules in VIATRA	The static aspects of UML models into stochastic Petri Nets, Transforming UML Statecharts into Extended Hierarchical Automaton
(d)	K.Lano, S.K.Rahimi [LR10]	Specification of model transformation using constraints and their verification	UML-RSDS for model transformation	Model transformation of UML into relational database schema
(e)	S. Weißleder [Wei09]	Semantic-preserving test model transformation for interchangeable coverage criteria	Formal framework for coverage criteria	Model transformation of State machine into System Under Test (SUT)
(e)	V.Chimisliu, C.Schwarzl, B.Peischl [CSP09]	Semantic preserving model transformation, which allows coverage-based test case generation	The description rules are defined with Extended Backus Nauer Form, the behavioural domain is an input-output labelled transition system	Model transformation of UML Statecharts into LOTOS

2.3.2 Discussion

The idea to ensure semantics preservation for a model transformation between models of different types is not new. Approach [GGL$^+$06] presents a mechanised proof of semantics preservation. The syntax of modelling languages within the approach is specified by means of graph transformations. The specification is later translated into Isabelle/HOL. Unfortunately, the behavioural semantics is not given as a certain formalism, but implemented in Isabelle/HOL directly. The question arises if it is possible to transform correctly a given behavioural semantics into the Isabelle/HOL notation. The approach also misses explicit definition of the version of bisimularity, therefore a class of preserved properties is hard to define. The proof itself faced some problems since it was not trivial to represent graph transformations within Isabelle/HOL. Approach [Str08] explains more about not trivial definition of a match within Isabelle/HOL. Approach [GGL$^+$06] was illustrated on an example of SDL automata and PLC-code. The trivial semantics of source and target languages throws a doubt about applicability of the method to a general case.

Another approach [EK04, HKR$^+$10a] to ensure semantics preservation between any given model of a source language and a resulting model of a target language was developed in parallel with this thesis. The approach relies on modelling languages specified by means of graph transformations and involves the technique called *borrowed context*, that is different to the standard application of graph transformations. The method requires an additional theory on top of the knowledge about graph transformations and tricky changes of the original specification for a behaviour of the models.

There is a lot of work done in the area of program checking [Gle03, NR01, NL97, CCN06, CEI$^+$05, MCBE06, KH08]. In the context of this area the source model is a program, the compiler plays the role of a model transformation [AU77]. The idea is to show that the program written in higher programming languages is correctly translated into native machine code. The modern methods for correctness of the compiler specification involve automated theorem provers [HV91], that brings the compilers to a considerable confidential level. However, the methods are hardly implemented for abstract modelling languages, because of the big gap between modern modelling languages, e.g. UML, and machine code.

A special area of behavioural semantics preservation during model transformation is refactoring, when there is a need for re-implementing existing functionality with a slightly different notation. There are some interesting approaches, for example [NK08]. It gives an interesting idea for using graph transformations for a model transformation, which involves using cross-links in order to show the trace equivalence. The cross-links are special associations to link the elements of a source model to the elements of a target model. The relation over the models is built during the transformation and then it could be traced using the associations.

There are many other approaches in refactoring area [vKCKB05, MTR05, GSMD03, RW07], however we present only some examples, mainly when graph transformations are used. We included the example from the area of logic programming, when approach [PP91] ensures that the transformation rules preserve the semantics for Prolog programs, but the method is based on a concrete example of the Prolog language and it is not clear how to extend it to modelling languages.

The problem of behavioural properties preservation could be solved by verifying some concrete source and target models. For example, approach [Var04] assures the syntactical correctness of the transformed models and propose to use a method of model checking for verification of behavioural properties.

Testing of model transformation [Wei09, CSP09, SCDP07, CDSS02] is based on the coverage criteria. They normally refer to a metric, which will give an idea about how well the system was exercised by some test cases. Unfortunately, testing allows only to check a finite number of cases and fails to ensure behavioural semantics preservation in general.

In our survey we considered only approaches, which involve a preservation of behavioural semantics during model transformation. There are different types of semantics, i.e. denotational [Gor79], institutional and so on. The algebraic techniques provide a tooling to define meta-models and the institutional semantics [BM10], which specifies the satisfaction relation between models and sentences. Approach [BKMW09] shows the institutional semantics preservation during model transformation. There were also tries to connect algebraic techniques with graph grammars [BCM02, CFR08]. To show that some notions, as for example history preserving bisimulation, is also decidable for graph grammars. However, these approaches stay on a theoretical level and it is not straightforward how to apply them to the real languages.

2.3.3 Conclusion from the Survey

We can conclude from our survey that there is no existing standard approach to semantics preserving model transformation which completely meets our requirements. In particular, meeting the requirements understandability, as well as finding a general solution, are difficult to achieve.

2.4 Concept of our Method

Summarizing the interesting ideas from the survey we want to introduce the main concepts of our solution, which is a method that fits the idea of the MDA approach and satisfies the requirements specified in Section 2.2. The method consists of the following steps:

(I) A specification of modelling languages L_1 and L_2.
(II) A specification of model transformation MT over $M_1 \times M_2$.
(III) A method for showing that for *all* behavioural properties φ formalized within a language Σ and their interpretation $\chi(\varphi)$ the following statement holds:

$$M_1 \models \varphi \quad \Rightarrow \quad M_2 \models \chi(\varphi) \tag{2.2}$$

The definition of modelling languages is the essential part of model transformations. The ability to show properties preservation is closely connected with two factors: a formalism for modelling languages and the way a model transformation are specified. Therefore, we dedicate the first two parts of this section to a widely accepted standard for syntax definition, as well as fundamental concepts of behavioural semantics definition that we use in our solution, i.e. (I). After that, we discuss the ideas of our solution for (II) and (III).

"A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer."[MDA]

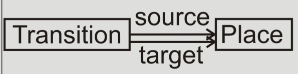
2.4.1 Syntax Definition

Syntax of the language defines the appearance and the structure of the sentences. It has nothing to do with the meaning of the structures which it defines. To describe syntax of a language formally means to describe formally *concrete* and *abstract* syntax.

The main difference is that concrete syntax defines precisely the syntactic structure according to some formal grammar, and abstract syntax in opposite avoids some details (for instance, grouping parentheses are implicit in the tree structure) in order to provide more abstract rules to describe the syntactical structure. The syntax is 'abstract' in the sense that it does not represent every detail that appears in the concrete syntax. In this thesis we work only with an abstract syntax.

The MDA approach suggests using a meta-model proposed by the Meta Object Facility (MOF) [MOF] and the Object Constraint Language (OCL) expressions [OMG] to define an abstract syntax. Prior to this notation, there were other modelling techniques, such as set theory, Backus-Naur-Form (BNF), natural languages to define a language syntax. These techniques made a significant effect in the development of the meta-model approach and the OCL notation. [MMAB$^+$08] provides a row of examples claiming that the traditional mathematical techniques may be equally expressed using meta-models and OCL. We shortly introduce some techniques in Table 2.2 on the example of the definition of an abstract syntax for one modelling language, called Petri nets [Rei85]. Natural language is one of the first modelling techniques to specify an abstract syntax. This technique uses descriptive rules, that prescribe how the sentences could be built. In set theory mathematical

Table 2.2: Example of abstract syntax definition for Petri nets with different languages

Language type	Language	Examples of the constructive rules	Definition by
Natural language	English	Places and transitions are connected by directed edges.	Descriptive rules
Mathematical language	Set theory	W:(S x T) ⋃ (T x S) → N	Mathematical formulas
Theory of programming	Extended Backus-Naur-Form	<statement> := <transitionStatement> | <placeStatement>	Context-free grammars
Visual language	MOF		Meta-model, OCL

formulas identify the structure of languages. BNF, that represents a set of derivation rules or grammars [Cho57], is used to describe a syntax of languages from programming theory. These three described techniques are predominantly text based. The meta-model notation is in opposite designed for visual languages.

"In BNF, the syntax is presented in an entirely text based format and although complete and theoretically fit for purpose, it presents a possible conceptual barrier to the ease of understanding for a typical human reader. Further, BNF is overly specific regarding the nature of the syntax whereas the graphical based format of UML primarily introduces the abstract concepts in an easily accessible and pictorial manner. Recent works such as [WK05, AP04] explore the relationships between BNF based definitions of syntax and metamodels."[MMAB+08]

The meta-modelling concept is currently highly accepted for the definition of abstract syntax of modelling languages. The meta-modelling notation resembles the notation of UML Class Diagram [UMLb] a lot, however it is mistakenly to believe they define the same thing, because the meta-model standard is prescribed for a definition of abstract syntax and a class diagram can specify a concrete syntax too.

The meta-model approach is beneficial when it comes to definition of complex modelling languages, consisting of several individual models. In comparison for example to Backus-Naur-Form, when introducing a new element or combining two different models into one requires a lot of changes and considerable extend of original definitions, the meta-model approach allows easily model the introduced operations into one meta-model. This feature is particularly useful for the specification of model transformations between two different languages. The approach allows to model completely different structures in one model.

Further, we explain the meta-model in details. The MOF-based meta-model approach has a layered structure. Each layer specifies the level of model abstraction. Figure 2.5 provides an overview of the OMG four-layer structure. The example is provided for a Petri net partial specification and illustrates the main concepts. On the topmost layer (M3) the MOF is located. It provides the construction elements and rules that could be used for the definition of a language is meta-model. The example of such constructs is an element `Class`. The usage of the element `Class` is displayed on the UML layer (M2), where a fragment for the definition of the UML meta-model is presented. There are three elements `Attribute`, `Class` and `Instance` that supposed to be the constructs for the bottom layer. The element `Class` on the meta-model layer is different to the MOF element `Class` and is connected with it by `«instanceOf»` relation, that could cause misunderstanding due to the instance issues within the one layer and within different layers. The user model layer (M1) expresses the elements of the problem domain. The bottom layer (M0) contains the objects that model supposes to represent. In our example there are the run-time elements.

In spite of all mentioned benefits and a big acceptance, the meta-model approach has some shortcomings, too. In the following, we discuss them and bypass solutions shortly, in order to avoid misunderstanding and introduce the proposed solutions to the user. One of the drawbacks is connected with an instantiation of the meta-model. The four-layer construction could be hardly understood due to the similar constructions of the bottom three layers of the hierarchy. The concept of instantiation through the layers and instantiation on the `User model` layer could cause misunderstanding. There are some suggestions, that are used in this thesis to handle the problem of instantiation.

- The MOF suggests to use meta-model concept with “as few as 2 levels and as many levels as users define”.
- The approach [AK03] separates the dimension of meta-modelling, giving rise to two distinct forms of instantiation. One dimension is concerned with language definition and makes use of *linguistic instantiation*. The other dimension is concerned with the domain definition and thus uses *ontological instantiation*. The linguistic instantiation is used by the modelling tools builders. The ontological instantiation is used by domain experts. However, the MOF language and its derivatives recognize only the linguistic instantiation. The

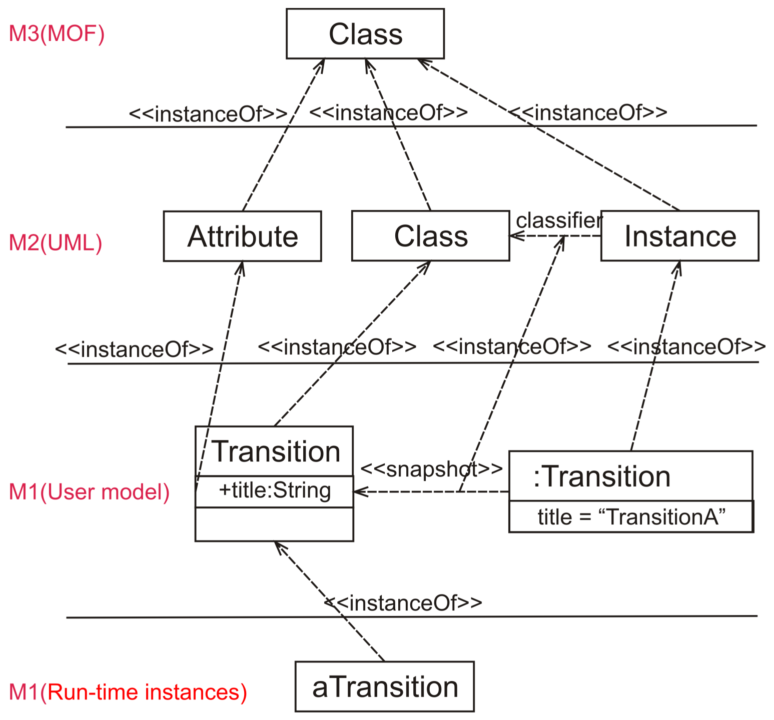


Figure 2.5: Illustration of the four layer structure of the UML/MOF framework (reproduced from [OMG])

ontological `instance-of` relation may be defined, but it would be just an ordinary UML association.

- The research [AK01] proposes a row of methods for modelling the multiple modeling layers.

Example 2.4.1 (Abstract syntax for the Petri nets language)**.** *In the following, we present an example of an abstract syntax definition of the Petri nets language. Instead of representing the four layer structure (the M2 and the M3 layers are the same as in Figure 2.5), we introduce only a meta-model.*

Syntax of Petri nets is originally defined as a tuple $N = \langle S, T, I \rangle$, where S is a nonempty finite set of places, $T \subseteq N_+^S \times Act \times N_+^S$ is a finite set of transitions, N_+^S denotes a multiset of places including at least one element, Act is a set of labels, $I \subseteq S$ is a set of initial places.

To implement such a structure with a meta-model, we do the following. For each set we define a separate element type: Place*-type element for set S,* Transition*-type element for set T,* Initial*-type element for set I. Since transitions are defined as a product over set S and set of labels Act, we specify an attribute for a* Transition*-type element, which value corresponds to an element from Act, and we use the associations of types* source *and* target *to implement the mapping function $\times$ of the* *product. To specify that set I is a subset of S in the type graph, we use an association of type* Initial. *We assume that each* Transition*-type element is always connected to at least one* Place*-type element with a* source *edge and to at least one* Place*-node with a* target*-edge. Therefore, each* Transition*-type element has at least one input* Place*-type element and at least one output* Place*-type element. If a* Place*-node is connected to an* Initial*-type element, it means that the place is from the set I. We present a type graph of Petri nets in Figure 2.6 in the left.*

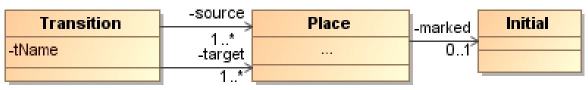


Figure 2.6: Meta-model that represents an abstract syntax model of Petri nets

We continue with constructing an ontological instance for the specified meta-model, that is a Petri net which describes a functionality of a simple vending machine. The vending machine can perform only several actions: it collects money, provides a choice between juice and water, delivers a chosen drink. The Petri net that describes the vending machine is an object diagram (see Figure 2.7 in the left).

The Petri net, which describes a structure of the vending machine, could be also illustrated differently. The representation is easily for human understanding, when the Petri net is depicted in the original graphical notation (see Figure 2.7 in the right). The dark rectangles represent Transition*-elements, which are labelled with the*

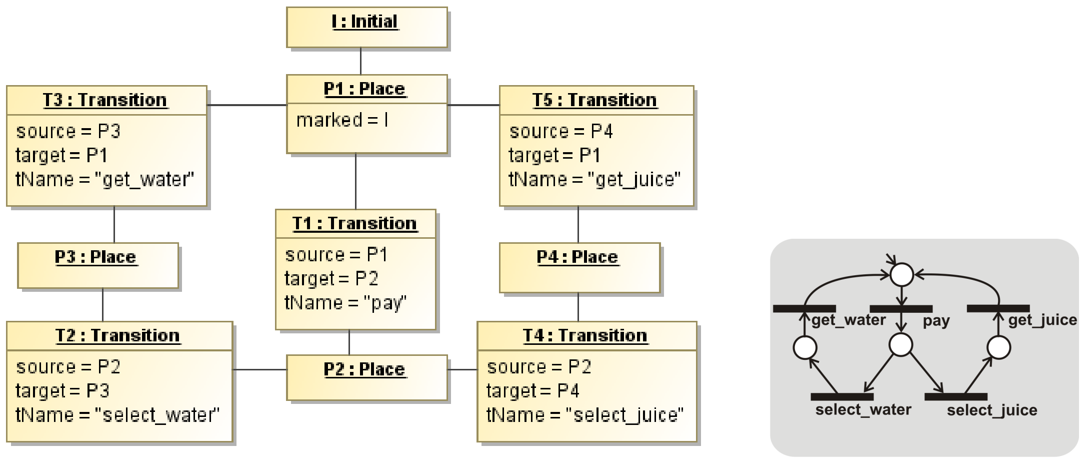


Figure 2.7: Instance of the meta-model, which describes the vending machine, in the left and the same Petri net in the original graphical notation in the right

correspondent actions that the vending machine can perform. The circles represent Place-*elements. The initial place is marked with an edge that has no source. The directed arcs describe which places are pre- and/or postconditions places for which transitions (signified by arrows).*

In this thesis we do not use the meta-modelling approach directly. Instead, we implement the meta-modelling concept with a help of graphs. Graphs is a convenient tool, because meta-models, i.e. class diagrams, could be treated as labelled graphs [BHM09, Roz97]. Then, the graph transformation rules could be used for the specification of a model transformation [SK08]. This approach is a common one for a definition of model transformation [FKS07] and has a number of advantages. First of all, the graph representation brings the research to another level of abstraction. The expression of models with a formal notation allows to study the structure of models with a theoretical approach. Besides, a graph is a promising tool with a strong fundamental background that allows to show additional features within a formal theory, in particular to analyse new properties of modelling languages. Moreover, due to the availability of GTR tools, a model transformation based on graph transformations is executable. The formal definition of graph transformation theory will be presented in Chapter 3.

Later we consider that a graph represents an abstract syntax of our language. The task of this graph is to specify the structure of other graphs, which are elements of the language (see Figure 2.8). We specify how one graph complies with the other in the next chapter.

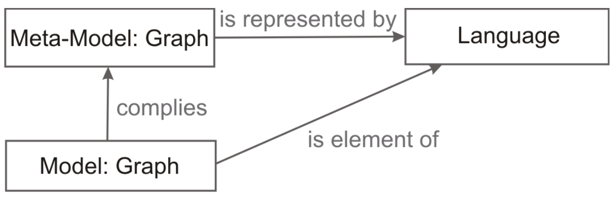


Figure 2.8: Main concepts of language structure defined with graphs

Summary: We require an abstract syntax of the languages L_1 and L_2 be definable with the meta-modelling approach. Abstract syntax defines the structure of a modelling language and is called “abstract” in the sense that it does not represent every detail that appears in concrete syntax. In this thesis, we do not work with a meta-model directly, instead we involve graph transformation theory that brings the research to another level of abstraction. Graph theory allows implementation the meta-modelling layer structure and other important concepts. Since we work only with abstract syntax in this thesis, under the word ‘syntax’ we assume abstract syntax.

2.4.2 Behavioural Semantics Definition

While the area of syntax is thoroughly studied and there exist the common standards for syntax definition, the area of semantics is not so well developed and the approaches broadly differ. There is still a big discussion and big confusion around the definition of semantics [har04, Mos01]. The reason for this is that semantic features are much more difficult to define and to describe.

There are many ways for semantics specification [BKMW09, BM10, SS71]. However, since we decided to define an abstract syntax with a graph, it is reasonable to specify the behavioural semantics by means of graph transformations.

The semantics based on graph transformation rule is related to *operational* semantics [Pad82, Plo04, Plo81, CHM00], that means that the rules describe how elements change. The rule consists of preconditions, which have to be met for a rule to apply, and postconditions that describe how the elements should be changed. The state in operational semantics involves abstract syntax structure. Each rule is supposed to transform one state into another.

The graph transformation rule bases on the same principle. It consists of two patterns: a pattern L for preconditions and a pattern R for postconditions. The condition, described with the pattern L, must *match* a graph structure. If the *match* succeeds, then the matched structure is replaced with the pattern R (see Figure 2.9). Graph transformation rules define the changes of graphs.

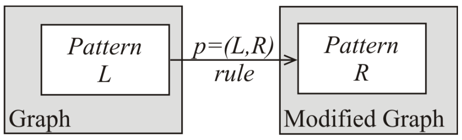


Figure 2.9: Graph transformation rule overview

Further, we explain how to specify the *semantic domain* by means of graphs. The graph responsible for an abstract syntax (or syntax graph) consists of static elements, which describe the structure, but not behaviour. The syntax graph could be extended with dynamic elements, which are related to the behaviour of a model. The extended graph is called *run-time graph* (see Figure 2.10), which specifies semantic domain. Graph transformation rules define the changes of the run-time graph. We say that graph transformation rules are *defined over* the run-time graph. However, the graph transformation rules are *applied* to the instances of the run-time graph.

Graph transformation rules compose a *graph transformation system*, which *fully* describes behavioural semantics (the language has no other behaviour except those, which is described within a graph transformation system). A result of the application of graph transformation rules from graph transformation system to a graph is a *Labelled Transition System* (LTS), which consists of states and transitions. The

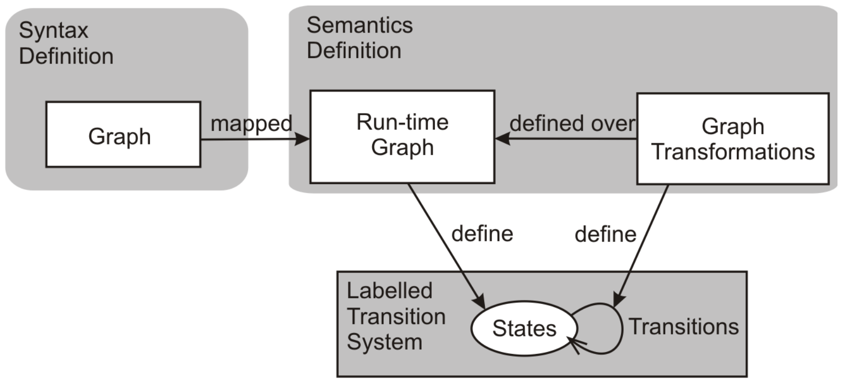


Figure 2.10: Behavioural semantics defined by graph transformations

graphs typed over the run-time graph are the states and graph transformation rules define the transitions, which are labelled with the names of rules (see Figure 2.10).

We have a special interest on an LTS, because an LTS is a perfect common domain for comparison of behavioural semantics of different modelling languages at abstract level. The comparison could be established by performing a binary relation between states, i.e. between rum-time graphs in our case. This topic will be discussed in detail in Chapter 4.

The idea to define a behavioural semantics for modelling languages by means of graph transformation is not new. For example, the Dynamic Meta-Modeling (DMM) approach [HHS01] involves the described technology. Jan Hendrik Hausmann in his thesis [Hau05] specified the method how to define behavioural semantics for visual languages and performed a case study for UML 2.0 Activities [UMLa]. Jochen Küster proposed a behavioural semantics by means of graph transformations for State Charts Diagrams [KÖ4, EHK01], as well as Sabine Kuske [Kus01]. Arend Rensink implemented the behavioural semantics by means of graph transformation for the TAAL language (a textual Java-like programming language) [KKR06].

Being very convenient for the application to visual and programming languages, the graph transformations are carefully studied within scientific communities from theoretical perspective [CHM00, CFR08]. Such questions as termination and specific properties of confluence were answered in [EEKR99]. [Cou09] presents the graph grammars expressed in specialized logics such as monadic second order logic. There are automated approaches based on model checking [Var02]. The verification of structural properties is the main focus of [Str08]. The are many others interesting examples, which let the impression that the graph transformation tool is in the interest of studies and a well-accepted mechanism.

Example 2.4.2 (Behavioural semantics definition for the Petri nets language)**.** *We return to our example of the Perti net language and give an idea, what it means to define a behavioural semantics by means of graph transformations. Since we did*

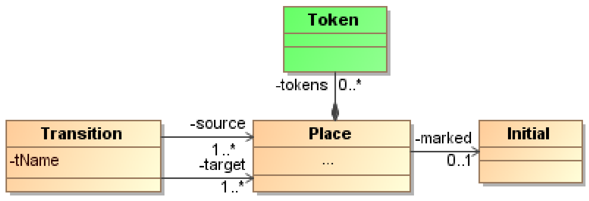


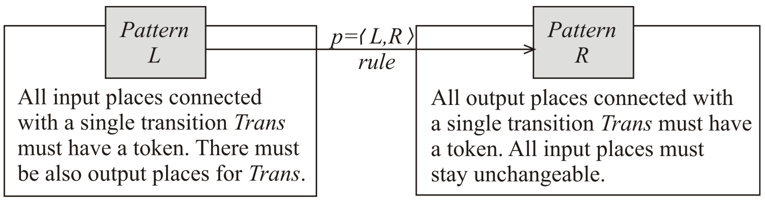
Figure 2.11: Run-time meta-model for Petri nets

not define graphs and graph transformation rules formally, our example gives a very brief overview. We use a meta-model notation for a run-time graph and our pattern (Figure 2.9) for a graph transformation rule.

The behaviour of Petri net is defined by a token flow, *which could be placed only at places. A place marked as* initial *indicates the presence of tokens before graph transformation rules were applied. Depending on the direction of arcs (source and target labels of binding edges) there are* input *and* output *places for a transition. A transition may* fire *tokens, if there is a token in each input place, then the token are copied in each output place and removed from input places.*

To define a semantic domain for Petri nets, we firstly extend a meta-model from Figure 2.6 with Token-class, which has an association with Place-class. See the result on Figure 2.11, the element, which describes the behaviour, is colored as green.

We present a graph transformation rule, denoted as pnMoveToken, *in a textual form. It consists of two patterns* pnMoveToken $= \langle L, R \rangle$*. The first pattern* L *defines the conditions which must hold for a graph in order the rule has a match. It says that there exist a transition* $Trans$*, such that all input places for a Trans have token and there must be at least one output place. The pattern* R *defines how the pattern* L *must look like after the rule application. It says that all output places connected with a single transition Trans must a have token. The input places stay without changes. It says nothing about tokens for output places, that means that they must be removed by the rule.*



All input places connected with a single transition *Trans* must have a token. There must be also output places for *Trans*.

All output places connected with a single transition *Trans* must have a token. All input places must stay unchangeable.

Figure 2.12: Textual interpretation of the graph transformation rule $MoveToken$

If we consider the already examined instance, which an assumption that one place is marked as initial (see Figure 2.7 in the left), then after step-wise application of

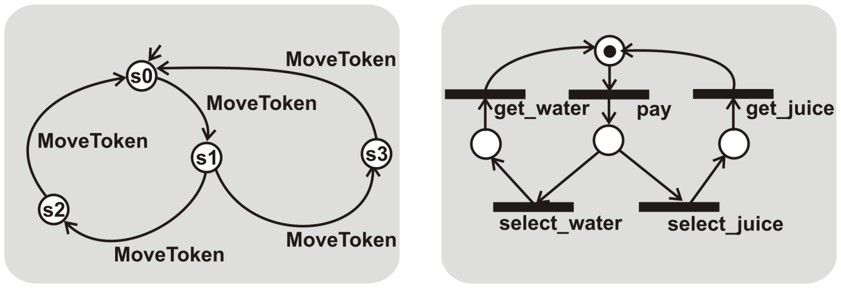


Figure 2.13: Example of a labelled transition system in the left and a Petri net from the state s_0 in the right

our rule to this instance, we can generate an LTS. It consists of states ($s0$*,* $s1$*,* $s2$*,* $s3$*), which are graphs. For example, a corresponding Petri net for a graph from the state* $s0$ *is depicted in Figure 2.13 in the right. The transitions in the LTS are labelled with the name of the graph transformation rule name.*

Summary: In our approach we require the languages L_1 and L_2 be equipped with behavioural semantics specified by means of graph transformation rules. The main advantage of such a specification is the possibility to generate an LTS. Then two different languages could be compared through their behavioural semantic models, i.e. LTSs.

2.4.3 Semantics Preserving Model Transformation

Now we introduce the main concepts of our solution for specification of model transformation and ensuring its correctness.

We consider a model as a graph and therefore specify our model transformation by means of graph transformations, too. Thereof, we use graph transformations in this thesis twice: once for behavioural semantics definition and then for a model transformation specification. The scenario of our model transformation is explained in Figure 2.14. The languages are defined by meta-models, which describe the instances. The model transformation is specified with respect to the meta-models. The specification of model transformation is implemented by a transformation engine, which takes a source model as an input and returns a target model. There are some remarks for the scenario:

- A model transformation specified with respect to the meta-models guarantees transformation be applicable to every model of the source language.
- The usage of graph transformation tools makes our model transformation understandable, which increase the chances of widespread adoption. In addition, the model transformation is both formally defined and executable.

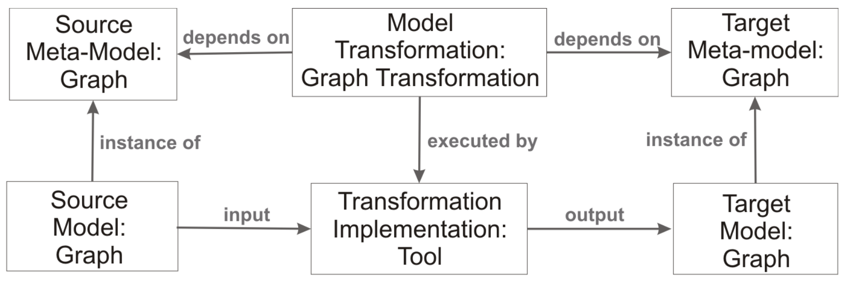


Figure 2.14: Idea for a model transformation implementation

- Our transformation is also bidirectional, which means that it transforms the source model into a target model, and the inverse transformation is also possible. The important idea is to keep correspondences between the equivalent elements until the end of transformations. This will help us prove behavioural preservation.

We deal with languages whose behavioural semantics could be formally specified by means of a graph transformation system. As mentioned in the previous subsection the graph transformation system gives rise to an LTS, in which transitions represent applications of graph transformation rules (see Figure 2.15). Then, an LTS could be considered as the underlying common domain for the source and target languages. On the generated LTS we can compare the behaviour of the source and target languages.

However, there is an obvious problem: the labels in the LTSs generated by different graph transformation systems are not the same (transitions in LTSs are labelled with graph transformation rule names from different systems). To solve this problem, we analyse the graph transformation systems and map equivalent rules to each other, i.e. rules, which perform similar behaviour. Then, the corresponding labels of LTSs which are mapped become observable, in the sense that it is possible to observe the actions from a point of view from the opposite LTS. All other labels become invisible. After the labels of LTSs are mapped to a common domain, it is possible to establish an equivalence relation over the LTS and show that the ordering of corresponding methods is the same.

We want to establish an equivalence relation for *any* instance of a source meta-model. This means that the proof needs to be performed once and then the correctness of model transformation for each particular instance is guaranteed.

Remind that our original goal is to show the preservation of behavioural properties. This can be formalized in the following way. If $Q(G_1)$ is an LTS generated for a source graph G_1, and $Q(G_2)$ is an LTS generated for a source graph G_2, then for every property φ and its interpretation χ for the target language the following statement holds:

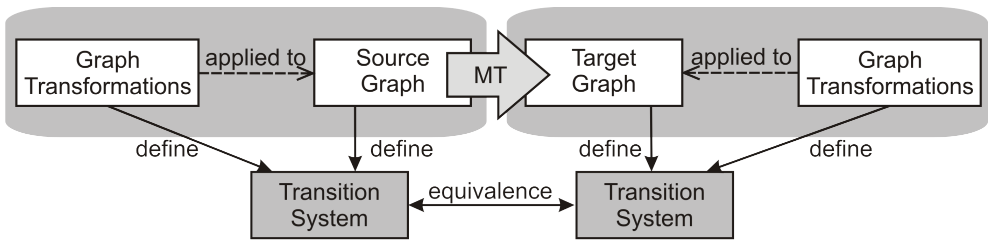


Figure 2.15: Solution idea for behavioural preserving Model Transformation (MT)

$$Q(G_1) \models \varphi \quad \Rightarrow \quad Q(G_2) \models \chi(\varphi) \tag{2.3}$$

There are different languages for specifying properties φ for LTS [Hol95, Sti95, Lar88], however the language with temporal operators, such as CTL* (Computation Tree Logic) [BCG88] is one of the most expressive [NFGR93] due to its ability to deal with notions such as *necessity*, *possibility*, *eventuality*, etc. CTL* has been recognized as a suitable formalism for specifying properties of concurrent systems [EH86, BAPM83]. In this thesis we study the connections between equivalence relation over the LTSs and the preservation of properties specified with action-based version of CTL*.

Among different equivalence relations over the LTSs, we are interested in a *weak bisimulation* [San95]. The reason for this is that it firstly allows to take into account the invisible steps. Secondly, it is possible to establish a connection between weak bisimulation and CTL* properties. By establishing such a connection, the equivalence over the LTSs will imply Statement 2.3.

Weak bisimulation, which is defined over the instances of run-time models, allows to establish a connection between corresponding elements and run-time properties, which these elements exhibit. Therefore, in order to interpret the behavioural properties of the source language L_1 for the target language L_2, we use the weak bisimulation and correspondences, generated by a model transformation.

2.5 Summary

In this chapter we defined the requirements for semantics preserving model transformation within the MDA approach. We presented a survey of related work and concluded that there is no existing approach that completely meets our criteria. Then, we described our idea for the solution, which is a method that consists of a language specification, a model transformation specification itself and a method to ensure behavioural semantics preservation.

CHAPTER 3

Foundations of Graph Transformations

Graph transformation theory is the basis of our approach, because we use graphs for syntax definition of a modelling language and graph transformations for two purposes: specification of behavioural semantics and model transformation (see Figure 3.1).

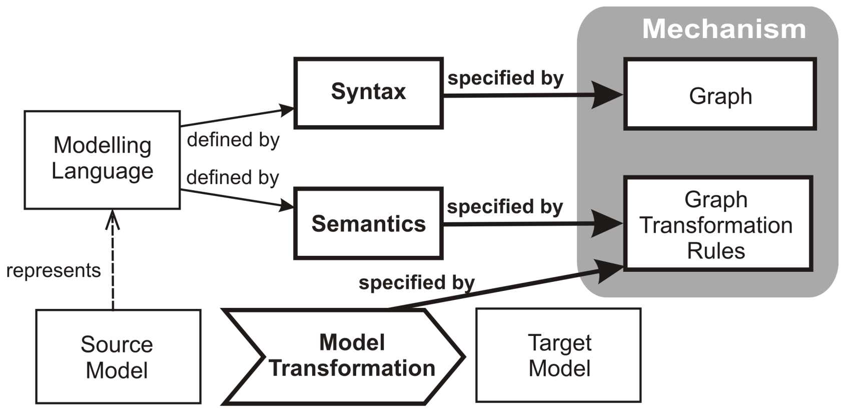


Figure 3.1: Discussion topic of this chapter

In graph transformation theory, a meta-model (which represents the syntax of a modelling language) and its instances are considered as graphs. A graph consists of nodes and edges, where a node stands for a class or an object and edges for connections between them. A typing concept allows to specify meta-models and their instances.

Graph transformation rules describe changes of a graph G by specifying pre- and postconditions. A precondition is a graph L, which must have a match to some subgraph G_1 of the original graph G. A postcondition is specified as a graph R, which replaces the subgraph G_1 in G. Therefore, behaviour of a model can be formalized as a set of graph transformation rules being applied to the graph that represents this model.

A model transformation can be also specified with graph transformations. In this case, a graph transformation rule formalizes the process of graph construction (syntactical part). The triple graph grammar technique used in this thesis allows to create the structure of a source graph and the corresponding structure of a target graph simultaneously. Moreover, this technique allows to keep correspondences between the corresponding nodes of source and target models, that is used later for ensuring behavioural correctness of the model transformation.

Graphs are an expressive, visual and mathematically precise formalism for modelling. Easily applied in area of visual modelling, the technique of graph transformations received the recognition of a comprehensive and reliable framework.

We proceed as follows. We turn to the main theoretical notions of graph in Sections 3.1. In Section 3.2, we explain the compatibility of the typing concept with the MOF concept. We provide a theoretical notion of graph transformations in Section 3.3. Specification of behavioural semantics by means of graph transformations is explained in Section 3.4. Section 3.5 introduces a triple graph grammar technique for a model transformation specification. The graph transformation tool used for implementation of graph transformations in this thesis is shown in Section 3.6.

3.1 Graphs

3.1.1 Graphs and Typed Graphs

Definition 2 (Graph)**.** A graph is a tuple $G = \langle V, E, src, tgt, lab_V, lab_E \rangle$ with a nonempty set of labels Sym, where V is a finite set of nodes, E a finite set of edges, $src, tgt \colon E \to V$ are source and target functions defined for each edge from the set E, $lab_V \colon V \to \mathsf{Sym}_V$ is a vertex (node) labelling function and $lab_E \colon E \to \mathsf{Sym}_E$ is an edge labelling function. We always assume $V \cap E = \emptyset$.

For a given graph G, we use V_G, E_G etc. to denote its components. Note that there is a straightforward (component-wise) definition of union and intersection over graphs, with respect to their components.

We assume that the source, target or labelling functions are consistent. This means that a function always maps a particular argument to the same value.

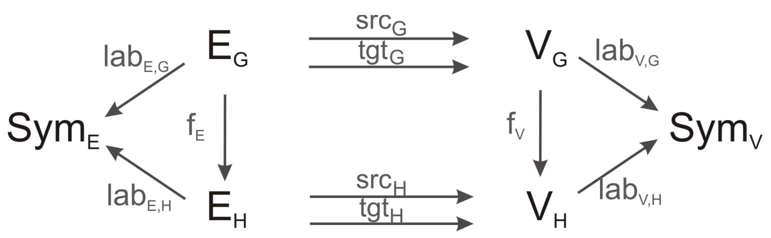


Figure 3.2: Morphism functions

Now we need to know how the graphs could be related to each other. For this purpose we define morphisms as a structure-preserving mapping between graphs.

Definition 3 (Morphism)**.** Given two graphs G, H, a morphism $f\colon G \to H$ is a pair of functions $(f_V\colon V_G \to V_H, f_E\colon E_G \to E_H)$ from the nodes and edges of G to those of H, respectively, which are consistent with respect to the source and target functions of G and H in the sense that $src_H \circ f_E = f_V \circ src_G$, $tgt_H \circ f_E = f_V \circ tgt_G$, $lab_{V,H} \circ f_V = lab_{V,G}$, $lab_{E,H} \circ f_E = lab_{E,G}$ (Figure 3.2). If both f_V and f_E are injective (bijective), we call f injective (bijective).

A bijective morphism is often called an *isomorphism*. Therefore, if there exists an isomorphism from G to H, we call them *isomorphic*.

The next step is to define how the graphs could be structured. A frequently used notion of graph structuring is obtained by *typing* graphs over a fixed *type graph*.

Definition 4 (Typing)**.** Given two graphs G, T, G is said to be *typed over T* if there exists a *morphism* $t\colon G \to T$.

A *typed graph* is a graph G together with such a morphism.

Given two graphs G, H typed over the same type graph (using morphisms t_G and t_H), a *typed graph morphism* $f\colon G \to H$ is a morphism that preserves the typing; i.e., such that $t_G = t_H \circ f$.

Example 3.1.1 (Type and typed graphs)**.** *We provide a small example of a type graph T and a typed graph G. Figure 3.3 depicts the graphical representation of graph. We describe the graph textually. Consider the following type graph* $T = \langle V_T, E_T, src_T, tgt_T, lab_{V,T}, lab_{E,T}\rangle$ *with:*

$$V_T = \{v1, v2\}, \qquad E_T = \{e1\},$$

$$src_T : e1 \mapsto v1, \qquad tgt_T : e1 \mapsto v2,$$

$$lab_{E,T} : e1 \mapsto X, \qquad lab_{V,T} : v1 \mapsto A, \qquad lab_{V,T} : v2 \mapsto B.$$

As you see, the textual definition of a graph takes much more space and attention then graphical definition. We shorten the textual definition of a typed graph

$G = \langle V_G, E_G, src_G, tgt_G, lab_{V,G}, lab_{E,G}\rangle$, *because it is not the main part of the example.*

$$V_G = \{v3, v4, v5\}, \qquad E_G = \{e4, e5, e6\},$$

$$\ldots\ldots$$

$$lab_{E,G} : e4 \mapsto x1, \qquad lab_{E,G} : e5 \mapsto x2, \qquad lab_{E,G} : e6 \mapsto x3.$$

$$lab_{V,G} : v3 \mapsto a1, \qquad lab_{V,G} : v4 \mapsto a2, \qquad lab_{V,G} : v5 \mapsto b1.$$

The goal of the example is to illustrate that it is important to ensure that all nodes and edges of the graph G typed over the graph T:

$$t_G : V_G \mapsto V_T, \qquad t_G : E_G \mapsto E_T,$$

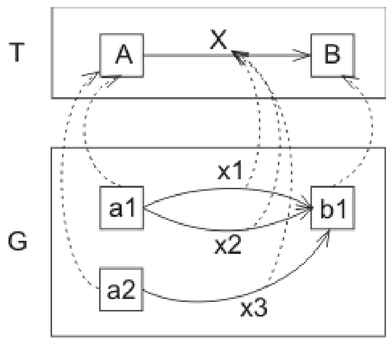


Figure 3.3: Example of a type graph T and a typed graph G

3.1.2 Type Restriction

Besides imposing some structural constraints over graphs, typing also provides an easy way to restrict to subgraphs:

Definition 5 (Type restriction)**.** Let T, U be graphs such that $U \subseteq T$, and let G be an arbitrary graph typed over T via $t : G \to T$. The *restriction of* G *to* U, denoted $\pi_U(G)$, is defined as the graph H such that

- $V_H = \{v \in V_G \mid \exists t(v) \in V_U\}$, $E_H = \{e \in E_G \mid \exists t(e) \in E_U\}$,
- $src_H = src_G \restriction_{E_H}$, $tgt_H = tgt_G \restriction_{E_H}$, $lab_{H,E} = lab_{G,E} \restriction_{E_H}$ and $lab_{H,V} = lab_{G,V} \restriction_{V_H}$.

Example 3.1.2 (Restriction)**.** *We consider the graphs T and G from the previous example and a graph U from Figure 3.4. The graph U consists of one node of the type A. We show the restriction of G to U. For this, we choice only those nodes and edges, which have a type node or edge in U. As a result we get a graph with two nodes labelled as $a1$ and $a2$ (see Figure 3.4).*

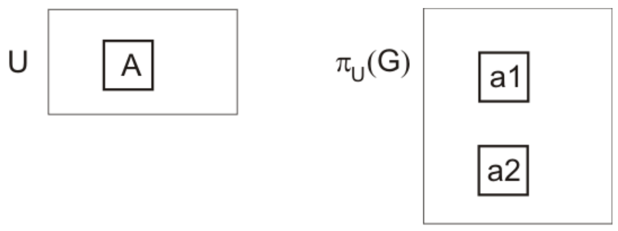


Figure 3.4: Example of a type graph U and a restriction of the typed graph G to U

3.1.3 Attributed Graphs

The graphs become a particularly expressive formalism when attributes are allowed. There is a number of approaches for attributed graphs [Kas06, EEPT06c, HKT02]. We do not go into detail, because the theory of attributed graphs demands a multitude of algebraic definitions and this is not our primary goal. Therefore, we provide only a very brief explanation in order to give an idea of how an attributed graph is defined, referring the reader to the related work [EEPT06a] for more details.

In order to support graph attribution we need to introduce *data type signatures*, which later could include information about types of attributes and operations on those types.

Definition 6 (Signature)**.** A *signature is a tuple* $SIG = \langle s_1, \ldots, s_n; op_1, \ldots, op_m\rangle$ *consists of sorts* s_i $(1 \leq i \leq n)$ *and operation symbols* op_j $(1 \leq j \leq m)$

We will define an attribute graph on the example of the following signature: $SIG = \langle int, +\rangle$, where there is only one operation of summation.

Below, we specify an algebra signature, as

$$DSIG = \langle int, +; arg0, arg1, result\rangle$$

where $+ : int \times int \times int$ is the sort representing the summation operation, and $arg0$, $arg1$, $result$ are projections. If we consider the tuples $\langle 1, 2, 3\rangle$ and $\langle 2, 5, 7\rangle$ then the projections for $DSIG$ would look as follows:

$$arg0(\langle 1,2,3\rangle) = 1 \quad arg1(\langle 1,2,3\rangle) = 2 \quad result(\langle 1,2,3\rangle) = 3$$

$$arg0(\langle 2,5,7\rangle) = 2 \quad arg1(\langle 2,5,7\rangle) = 5 \quad result(\langle 2,5,7\rangle) = 7$$

$DSIG$ is an algebra, because the properties of algebra, such as the existence of an identity element and inverse element and properties of associativity and commutativity hold.

We define an E-graph, in which we distinguish between two kinds of vertices, called graph and data vertices, and three different kinds of edges used for representation of E-graphs.

Definition 7 (E-graph)**.** An E-graph $G = \langle V_1, V_2, E_1, E_2, E_3, src_j, tgt_j, lab_{V_i}, lab_{E_j} \rangle$, where $i = 1, 2$ and $j = 1, 2, 3$, V_1 and V_2 called graph and data nodes respectively, E_1 and E_2 called graph and node attribute respectively, E_3 defines the edges between data nodes,

source and target functions

- $src_1 : E_1 \to V_1, \quad src_2 : E_2 \to V_1, \quad src_3 : E_3 \to V_2,$
- $tgt_1 : E_1 \to V_1, \quad tgt_2 : E_2 \to V_2, \quad tgt_3 : E_3 \to V_2.$

labelling functions:

- $lab_{V_1} : V_1 \to \mathsf{Sym}_1, \quad lab_{V_2} : V_2 \to \mathsf{Sym}_2,$
- $lab_{E_1} : E_1 \to \mathsf{Sym}_3, \quad lab_{E_2} : E_2 \to \mathsf{Sym}_4, \quad lab_{E_3} : E_3 \to \mathsf{Sym}_5.$

An E-graph morphism $f : G_1 \to G_2$ is a tuple $\langle f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2} \rangle$ with $f_{V_i} : G_{1,V_i} \to G_{2,V_i}$ and $f_{E_i} : G_{1,E_i} \to G_{2,E_i}$ for $i = 1, 2$ such that f commutes with all sources and target functions.

Components of an E-graph with an additional explanation are illustrated in Figure 3.5.

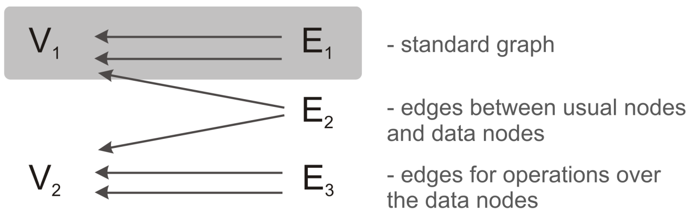

V1
E1
- standard graph
E2
- edges between usual nodes and data nodes
V2
E3
- edges for operations over the data nodes

Figure 3.5: Components of an E-graph

In the following, the graphs are replaced by E-graphs in order to allow node attribution. An attributed graph is then a structure, which consists of an E-graph and a set of values that represent all possible values available on attributes values according to some algebra.

Definition 8 (Attributed graph)**.** Consider a data structure $DSIG = \langle int, +; arg0, arg1, result \rangle$ and a E-graph G. An *attributed graph* $AG = \langle G, D \rangle$, where D is a $DSIG$-algebra such that $\uplus_{s \in S} D_s = dom(lab_{V_2})$, here S is a set of sorts of $DSIG$, and $dom(lab_{E_3}) = \{arg0, arg1, result\}$.

The data nodes are connected via edges from E_2 to normal nodes. For each data type s there is a domain D_s containing all possible values of that data type. Instance graphs will contain these values in nodes from V_2 and connect them to normal nodes. The structure from V_2 and E_3 could be considered as a *bipartite graph*, in which the labels for the nodes representing the instance of algebra operation $+$ (denoted also as *add*) form one set and the nodes representing the constant data values from the other disjoint set. The edges from E_3 have the same directions, namely from the set of algebra operations to the set of constant data values.

Figure 3.6 shows an example with a bipartite graph. The nodes with values 1, 2, 3, 5 and 7 are the nodes from the set V_2, they denote the attributes of some nodes from V_1 (the latter are not depicted in the Figure). The algebra operation *add* is represented as a separate node with three outgoing edges, which stand for two arguments and the result of the summation operation.

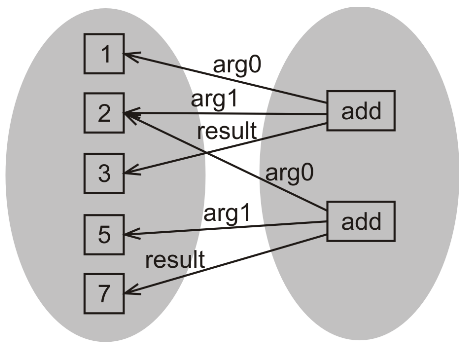


Figure 3.6: Bipartitional graph

In this thesis, in addition to $DSIG$, we use the following algebra signatures:

$$DSIG_{lt} = \langle int, <; arg0, arg1, result\rangle$$

where $<$: $int \times int \times boolean$ is the sort representing "less than" operation,

$$DSIG_{concat} = \langle string, *; arg0, arg1, result\rangle$$

where $* : string \times string \times string$ is the sort representing concatenation operation.

$$DSIG_{toString} = \langle int, string, toString; arg0, arg1, result\rangle$$

where $toString : int \times string$ is the sort representing an operation, which turns integer values into strings.

3.2 Graphs as a Tool for Syntax Definition

In this section, we explain how graph theory could be used for the description of abstract syntax. We claim that the MOF concept represented in Chapter 2 (Section 2.4.1) can be substituted with the typing concept.

A graph language represents language constructs such as graphs and edges. It is not tied to a specific application domain, but may be used for arbitrary modelling languages. In our case graph language uses directed, edge labelled graphs as underlying data model. Below, we discuss how these graph constructions could be used for the definition of an abstract syntax.

The *type graph* introduced in the previous section specifies the abstract syntax of a graph language. We denote the relation between a type graph and a graph language as “is represented by”. The task of a type graph is to express the specific *typed graphs*. Then a typed graph “complies” to a single type graph, where a type graph may have multiple compliant typed graphs. The task of the type graph is to specify the structure of typed graphs, which are elements of a graph language (see Figure 3.7). A graph language is a set of typed graphs, we denote the relation as “is element of”. The described definition is very similar to the MOF concept [SCF+05], if a type graph is considered a meta-model and a typed graph a model.

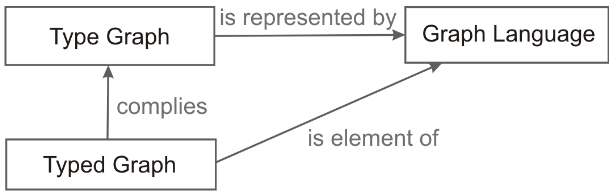


Figure 3.7: Relations between main concepts of graph language structure

Later we denote T^{st} a type graph, which describes the abstract syntax of a language. The index st stands for *static* in the sense that the graph T^{st} describes the static structure of a graph, which is not changed during the application of behavioural semantics.

Example 3.2.1 (Abstract syntax of Petri nets)**.** *We continue with the running example of Petri nets from the previous chapter. We define an abstract syntax with a type graph (the meta-model for Petri nets syntax was already introduced in Chapter 2 Section 2.4, therefore we only construct its equivalent graph structure here). See the result in Figure 3.8. All graphs of the Petri net graph language must be typed over the graph T^{st}_{PN}.*

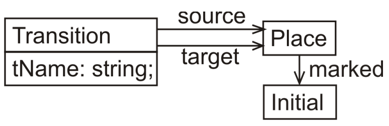


Figure 3.8: The type graph T^{st}_{PN}, which describes the abstract syntax of Petri nets

The typing concept could be used for a syntax description with a four layer structure of the MOF framework (see Chapter 2 Section 2.4.1). Here, we consider only three layers to define a graph structure with the MOF concept.

Figure 3.9 illustrates the idea of compatibility of the typing concept and the MOF. The layer structure is preserved, however models are defined by graphs. On

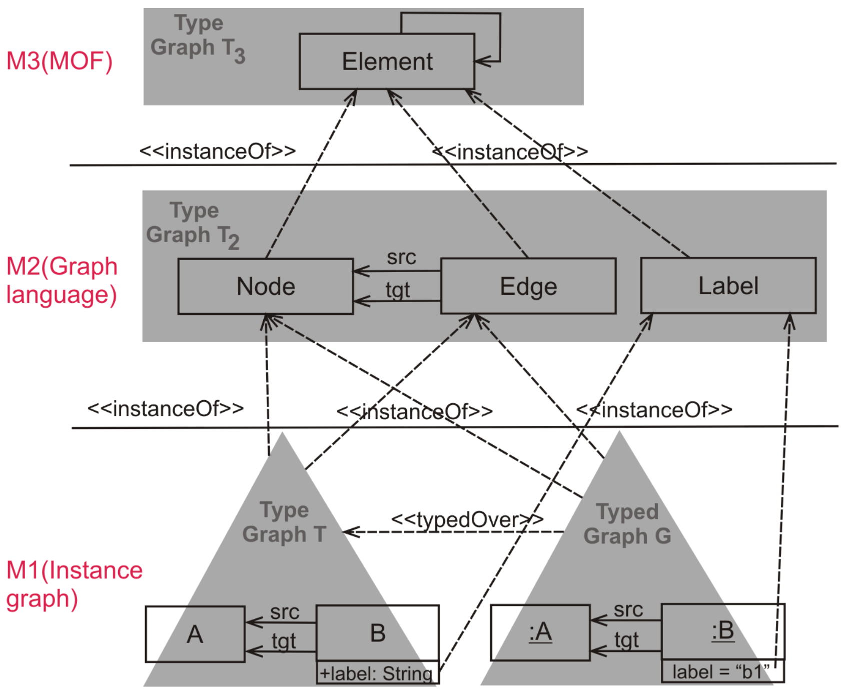


Figure 3.9: Illustration of a graph language definition with the three layer structure of UML/MOF framework

the top layer, there is a type graph T_3, which defines the `Element`. At the layer M2 we specify a type graph T_2, which includes the nodes `Node`, `Edge` and `Label`. The graph T_2 defines a graph structure in general. The type graph T_2 is in-turn typed over the type graph T_3.

Graphs in layer M1 define the concepts of a graph language. If the top two layers specify a part of abstract syntax of any graph language, a type graph T specifies a conceptual part of abstract syntax of a concrete language. We will latter use only graph T (or T^{st}) for the description of abstract syntax. The graph G is a typed graph, which is typed over T_2, or a linguistic instance of T. At the same time G complies T, and is an ontological instance of T (see [AGK09] for more details). In Figure 3.9 there are different relationship types to express elements of a typed graph. Each element of a typed graph is "typed over" a type graph T and is an "instance of" a type graph T_2.

After we specified syntax with a graph, we turn to the definition of behavioural semantics by graph transformations.

3.3 Graph Transformations

3.3.1 Introduction

The technique of graph transformations[1] is base of our method, because we use it for specification of behavioural semantics and for definition of model transformation. In this section, we provide important formal definitions.

To give a reader an idea of a graph transformation rule, we start with an example of well-known Chomsky grammars [Cho57], which has a lot in common with graph transformations. A grammar defines a formal language, which is a (usually infinite) set of finite-length sequences of symbols that may be constructed by applying production rules to another sequence of symbols. Symbols could be of two types: *terminals* (unchangeable symbol) and *nonterminals* (symbol to be changed). The initial sequence contains a nonterminal. A rule may be applied to a sequence of symbols by replacing nonterminals with those symbols that appear on the right-hand side. A sequence of rule applications is called a *derivation*. Such a grammar defines the formal language: all words consisting solely of terminal symbols which can be reached by a derivation from the start symbol.

The grammars below describe the construction of natural numbers. The language of this grammar is the set of all words, which a natural number can present. We use two nonterminals (`NaturalNumber` and `Digit`) and terminal symbols (‘0‘ . . . ‘9‘) for our definition:

```
NaturalNumber ::= NaturalNumber Digit | Digit
Digit ::= ‘0‘|‘1‘|‘2‘|‘3‘|‘4‘|‘5‘|‘6‘|‘7‘|‘8‘|‘9‘
```

Graph transformation has a similar concept [PR69]. Alike grammars, graph transformations describe the construction of graphs. However there is no distinguish between terminals and nonterminals. All possible patterns of graph components are nonterminals for a graph transformation rule.

> *“Similarly to Chomsky grammars, in the string case it is also possible to distinguish between terminal and nonterminal symbols for the production in grammars. In the case of nonterminals, the language would be restricted to graphs with terminal symbols only.”* [EEPT06a]

The graph transformation rule $p = \langle L, R, \mathcal{N} \rangle$ is a tuple of graphs L, R and $\mathcal{N}$ (see Figure 3.10). The graph L is a pattern which specifies preconditions. It is similar to the left-hand side part of a Chomsky grammar. If the pattern L was found in a graph, then it is replaced with a pattern R, which is by turn similar to the right-hand side of a Chomsky grammar. The pattern $\mathcal{N}$ specifies the *absence* of a structure and is an extension of left-hand side pattern L.

[1]The research area of graph transformation dates back to the 1970s. Today there are several topics of intense theoretical research, which overview is given in Volumes 1, 2 and 3 of the Handbook of Graph Grammars and Computing by Graph Transformations [Roz97, EEKR99, BCE^{+}99].

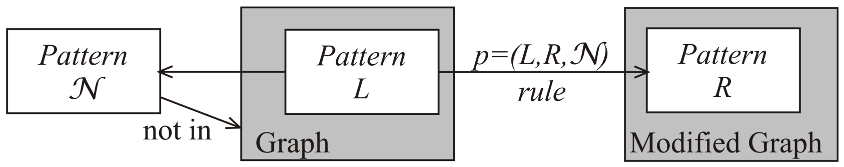


Figure 3.10: Rule-based modification of graphs

3.3.2 Basic Definitions for Graph Transformations

Further, we formally describe changes on graphs step by step. A graph transformation rule describes the change of a graph by means of a left-hand side, right-hand side of a rule (for marking the presence of certain structures) and negative application condition (to denote the absence of a structure).

Definition 9 (Graph transformation rule)**.** A *graph transformation rule is a tuple* $p = \langle L, R, \mathcal{N} \rangle$*, consisting of a left-hand side (LHS) graph* L*, a right-hand (RHS) graph* R *and a graph* $\mathcal{N}$*, which is a negative application condition (NAC).*

The elements in both left and right-hand side graphs are called *interface* of the rule: $I = L \cap R$ (in the example in Figure 3.11 the interface consists of the node labelled A).

In order to apply a graph transformation rule to a graph, we need to define a condition under the occurrence of LHS could be found in the start graph (also called the *host graph*). We say that a transformation rule $r = \langle L, R, \mathcal{N} \rangle$ is *applicable* to a graph G, if there exists an *injective match* $m\colon L \to G$ such that for no $N \in \mathcal{N}$ there exists a match $n\colon N \to G$ with $m = n\restriction_L$ (i.e., a negative application condition is *satisfied*), and moreover, the *dangling edge condition* holds.

Definition 10 (Dangling edge condition)**.** For all $e \in E_G$,

$$\mathit{src}(e) \in m(V_L \setminus V_I) \qquad \mathit{or} \qquad \mathit{tgt}(e) \in m(V_L \setminus V_I)$$

implies $e \in m(E_L \setminus V_I)$.

This condition can be understood by realising that the elements of G that are in $m(L)$, but not in $m(I)$ are scheduled to be deleted by the rule, whereas the elements in $m(I)$ are preserved. The condition then implies that if a node is deleted, then so are all its incident edges.

Given such a match m, the *application* of r to G is defined by extending m to $L \cup R$, by choosing distinct "fresh" nodes and edges (outside V_G and E_G, respectively) as images for $V_R \setminus V_L$ and $E_R \setminus E_L$ and adding those to G. This extension results in a morphism $\bar{m}\colon (L \cup R) \to C$ for some extended graph $C \supseteq G$. Now let H be given by

$$V_H = V_C \setminus m(V_L \setminus V_R), \qquad E_H = E_C \setminus m(E_L \setminus E_R)$$

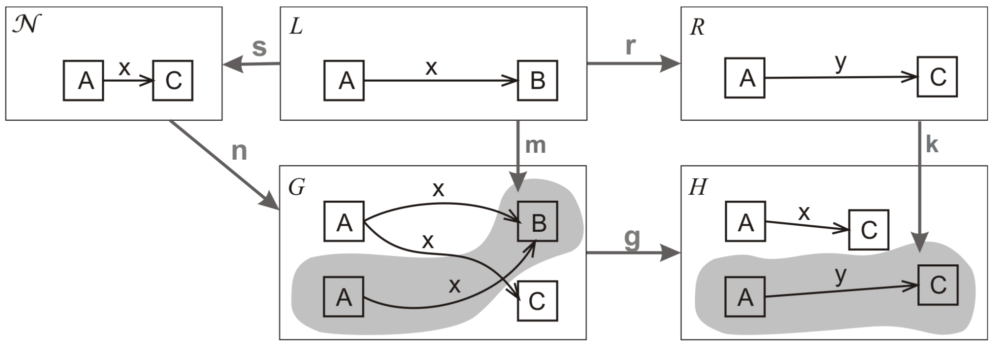


Figure 3.11: Basic graph transformation rule concept

together with the obvious restriction of src_C, tgt_C and lab_C to E_H. H is called the *target* of the rule application; we write $G \xrightarrow{r,m} H$ to denote that m is a valid match on host graph G, giving rise to target graph H, and $G \xrightarrow{r} H$ to denote that there is a match m such that $G \xrightarrow{r,m} H$. Note that H is not uniquely defined, due to the freedom in choosing the fresh images for $V_R \setminus V_L$ and $E_R \setminus E_L$; however, it is well-defined up to isomorphism. From now on we assume that the fresh images are chosen in some deterministic fashion, so that H is in fact fixed.

There are other definitions of graph transformation rules in the literature. The one used here is the one for *single-pushout* rewriting (SPO-rewriting) [LE90, Löw93, EHK$^+$97]. Another wide-spread approach is the *double-pushout approach* (DPO) [CMR$^+$97, Sch05]. The theoretical differences mainly relate to the handling of dangling edges (edges, for which either a source function of target function is not defined). If the rule application removes a node, but not an incoming or outgoing edge, then a question arises what happens with such edges. Under DPO approach, such a rule application is prohibited while SPO approach calls for an implicit deletion of the dangling edges. The example in Figure 3.11 demonstrates the difference. Under the DPO approach, the rule application would not be possible, because two edges which are labelled as x are not in the match. Under SPO, however it is implicitly deleted.

3.3.3 Injective and Non-injective Matches

Injective matching plays an important role in the process of a graph transformation rule application, because there are two different options for the conditions, under which a rule can be applied. Without mentioning these options, the rule can have two different interpretations.

In the case of injective match it is supposed that each element has a match to a different element in a host graph. In the case of a non-injective match, two separate nodes (of the same type) in an LHS pattern can be matched to one single node. For example, if the interface of a rule is as shown in Figure 3.12, the interface pattern has a match in case of non-injective match. If the property of a match is set as injective, then there is no match for the same interface pattern.

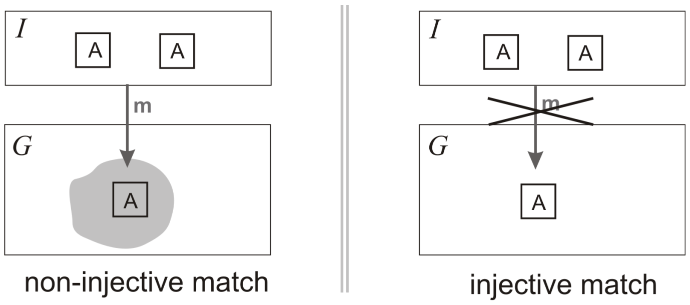


Figure 3.12: Difference between injective and non-injective matches

This must be taken into account during the definition of graph transformation rules, since they could appear different. To convert one rule into another, one can add explicit injectivity constrains (as NACs) to rules when necessary (see the Figure 3.13, "=" label for an edge means that the nodes must be matched to the same node).

In this thesis we use non-injective match.

3.3.4 Important Notation

Up to this point, we used the notation for a graph transformation rule presented by explicitly showing the LHS and RHS graphs. However, it is also possible to represent a graph transformation rule by merging them into one graph. Throughout the remainder of this thesis, we use the latter, one-graph approach. This implies that nodes and edges must be annotated according to their function within the rule. There are 4 types of elements:

- Nodes and edges which remain unchanged are depicted with black, solid, thin lines.

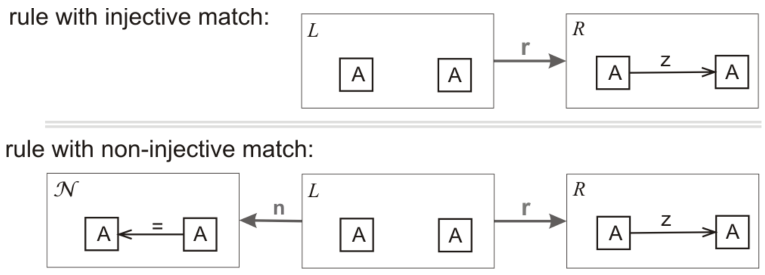


Figure 3.13: Sample rule with injective matching and modified rule of the solution with non-injective matching

- Nodes and edges created by the rule are depicted with green, solid, fat lines.
- Nodes and edges deleted by the rule are depicted with blue, dashed, thin lines.
- Nodes and edges which must not exist in the host graph for the rule to match are depicted with red, dashed, fat lines.

Figure 3.14 depicts the graph transformation rule from Figure 3.11 using the new notation.

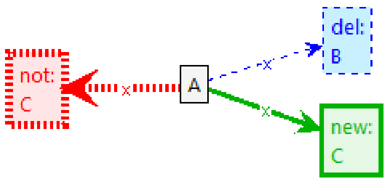


Figure 3.14: Notation for a graph transformation rule

3.3.5 Universal Quantification

Elements in the LHS pattern of a graph transformation rule have by default an existential quantification (∃), i.e. the elements in the pattern need to find *one* corresponding match in the host graph. However, there are situations in which universal quantification is necessary. For example, in the graph transformation rule for the behavioural semantics of Petri nets (see Chapter 2, Subsection 2.4.2) the LHS pattern must find all input places for the transition and assure that they have a token. For this, the Universal Quantification Structure (UQS) in the LHS of the rule must match *all* elements in the host graph, which fulfill the given constraints. In Figure 3.15, the UQS matches the Place-nodes, each one of them is connected with a Token-node and one single Transition-node. The host graph is shown in the original graphical Petri nets notation for simplicity.

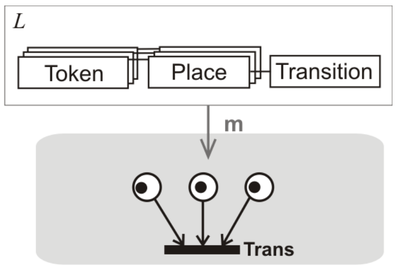


Figure 3.15: Example of the match of universally quantified elements

There are different approaches, which could be used to express UQS in graph transformations [Tae96, Sch91, JETE04]. We use the approach of Arend Rensink [Ren04b]. It studies graph predicates in which a graph is extended by several layers, each forming a logical negation of the one above it. The UQS structure is based on the logical statement that negation of existential quantification yields universal quantification. The example in Figure 3.16 specifies the condition that $\exists y : next(x, y) \vee \forall z : (next(x, z) \Rightarrow z = y)$.

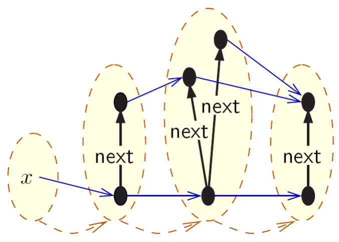


Figure 3.16: Example of a graph predicate (taken from [Ren04b])

We use special notation for the specification of universal and existential quantification, which consists of auxiliary nodes. These nodes are part of the rule and are connected using "in" – labelled edges. The quantifier nodes and in-edges must form a forest, i.e., a set of trees within a rule; in other words, it is not allowed that a quantifier node is "in" two distinct other quantifier nodes, or that there is a cycle of quantifier nodes. Moreover, existential and universal nodes must alternate, and the root nodes must be universal. In addition, there is always an implicit top-level existential node, with implicit in-edges from all the explicit (universal) root nodes.

The quantifier nodes are specified once more using special prefixes:

- forall ($\forall$): specifies a universal level, i.e. in a match of the entire rule, the sub-rule at such a level can be matched arbitrarily often (including zero times).
- exists ($\exists$): specifies an existential level, i.e. in every match of the entire rule, the sub-rule at such a level is matched exactly once.

The following is an example of universal and existential quantification structures (leaving out the actual rule).

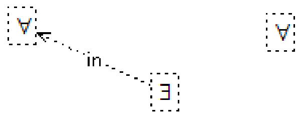


Figure 3.17: An example of a quantifier structure

The structure in Figure 3.17 corresponds to the quantifier structure of a predicate formula, where the branching stands either for conjunction (in the case of universal levels), or for disjunction (in the case of existential). The figure reflects the predicate structure:

$$\exists(\forall\exists \wedge \forall)$$

Every nesting level, represented by a quantifier node, contains a sub-rule. The containment relation is encoded by "at"-labelled edges from every node in the sub-rule to the corresponding quantifier node.

As a simple example, Rule (a) in Figure 3.18 will result in the removal of all Token-labelled nodes of all Place of a given (implicitly existential quantified) Transition. Rule (b) is a slightly more complicated variant, which picks exactly one Token of every Place that has at least one Token (for more information see User Manual for the Groove Tool Set [GRO]).

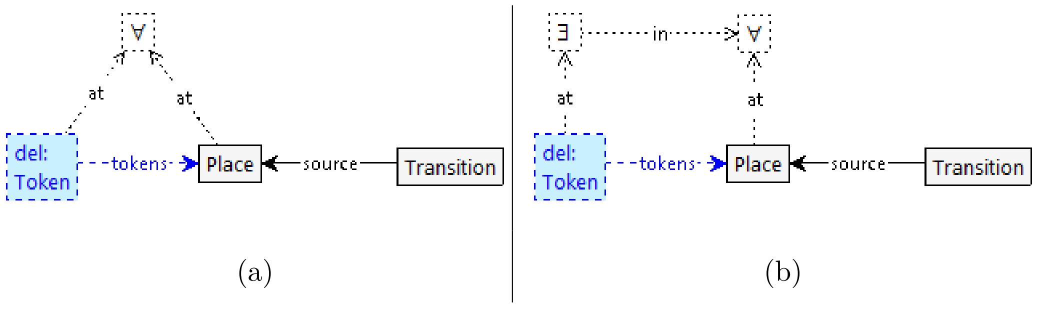


Figure 3.18: Illustration of two sample rules with a quantifier structure

3.3.6 Graph Transformations for Attributed Graphs

We have already explained the formalization of attributed graphs in Subsection 3.1.3. In this subsection we explain how we transform attributed graphs by means of an example, focusing on how to change attribute values. We are mainly inspired by [Kas06], where the method called *the method of signatures* is used to transform attributed graphs.

Let us assume that we have an attributed graph. It could be depicted in two ways. The first variant is the notation close to the standard UML notation (see Figure 3.19 in the left). The second variant is to illustrate it as a graph, in which each data node or attribute node is represented by a single node, labelled with its type of value and value itself (see Figure 3.19 in the right).

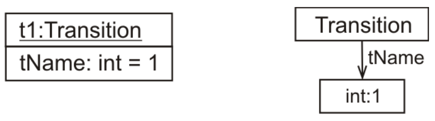


Figure 3.19: The UML notation for an attributed node (in the left) and a graphical representation for the same attributed node (in the right)

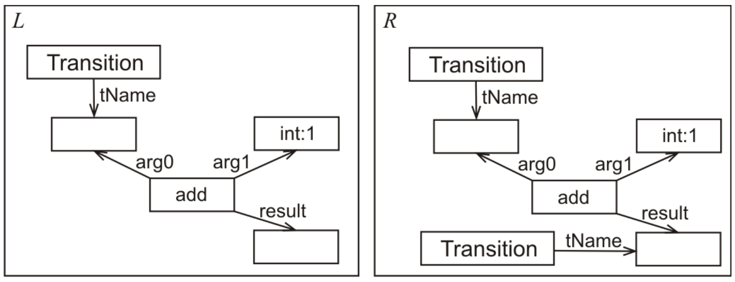


Figure 3.20: The rule for adding a new Transition-node with an integer value that is one time more than the value of some existing Transition-node

Specifying the transformation of attributed graphs basically consists of two parts: specifying (1) graph structure changes and (2) attribute value changes. The first part was already performed in Subsection 3.3.2. Here, we focus on the second part. A graph transformation rule for attributed graphs consists of auxiliary nodes: nodes representing the instance of an algebra operation and data nodes. Node that these nodes being in the interface of a rule (i.e. in $L \cap R$) must not have a match. Only in case of a data node is connected with a structure node, the data node must have a match to some node in a host graph.

Figure 3.20 shows a graph transformation rule, which creates a new Transition-node with an attribute, which integer value is to the one unit bigger than an attribute value of already existing Transition-node. The LHS pattern consists of a Transition-node, connected with a data node and the following auxiliary nodes: a nodes representing the instance of the algebra operation $+$, a data node with the integer value 1 and a data node that stands for the result of the algebra operation $+$. The required match consists of the Transition-node and its data value. The RHS pattern has additional Transition-node, which is connected with the data node that stands for the result of the algebra operation $+$.

3.4 Behavioural Semantics Based on Graph Transformations

In this thesis we use graph transformations to specify behavioural semantics of a modelling language. The syntax of a modelling language is specified as a graph T^{st}. A model of this modelling language is an element from a set of graphs typed over T^{st} (see Subsection 3.2). Then, in order to specify behaviour of the model, we extend the type graph T^{st} with additional elements, which attach a special meaning for static elements by being connected with them. The extended graph T^{rt} is considered a semantic domain of a modelling language. The graph transformation rules are specified over T^{rt} in a such way that do not affect elements typed over T^{st}, but only those nodes and edges, which specify a behaviour. The set of graph transformation rules which fully describe the behaviour of a system (it is assumed that there is no other behaviour except those, which is specified by these rules) form a graph *graph transformation system.* When applied to a graph, a graph transformation system gives rise to a *labelled transition system*, which summarizes all sequences of possible executions. A labelled transition system is considered as a behavioural model, which can be compared and analysed.

We continue with an example of definition of a run-time graph for the Petri nets language. We then formalize the graph transformation system. Finally, we formally define a labelled transition system.

Example 3.4.1 (Run-time type graph for Petri nets)**.** *Behaviour of Petri nets is defined by a token flow. An element called* token *flows through the net being fired by* transitions *and makes stops in* places*. To specify a run-time type graph for Petri nets, we extend a type graph from Example 3.2.1 with a* Token*-node, which can be connected with a* Place*-node from* T^{st}*. Figure 3.21 depicts the run-time graph* T^{rt} *for Petri nets.*

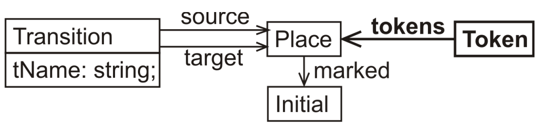


Figure 3.21: The run-time graph T^{rt} for Petri nets

The graph transformation rules form a graph transformation system, which defines the behavioural semantics of a language.

Definition 11 (Graph transformation system)**.** Let T^{rt} be a run-time graph. A *graph transformation system* is a partial mapping $\mathcal{RS}\colon \mathsf{Sym} \rightharpoonup \mathsf{Rule}$, where Rule is a set of graph transformation rules typed over T^{rt} and Sym is a universe of names.

Execution of a graph transformation system means a stepwise application of graph transformation rules from Rule to some typed (over T^{st}) graph. The graph transformation rule systems give rise to a labelled transition system summarizing all sequences of these executions.

Definition 12 (Labelled transition system)**.** A Labelled Transition System (LTS) is a structure $Q = \langle S, \rightarrow, \iota, L\rangle$, where S is a set of states and $\rightarrow\ \subseteq S \times L \times S$ is a set of transitions labelled over some set of labels L. Furthermore $\iota \in S$ is the start state.

The intuitive behaviour of LTS can be described as follows. The LTS starts in some initial state ι and evolves according to the transition $\rightarrow$. If s is the current

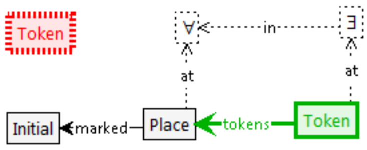


Figure 3.22: The pnInitial rule

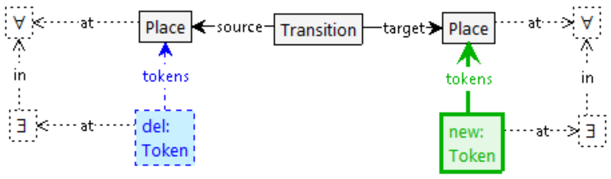


Figure 3.23: The pnMoveToken rule

state, then a transition $s \xrightarrow{\alpha} s'$ is selected *nondeterministically*. Then the action α is performed and the LTS evolves from the state s into the state s'. This selection procedure is repeated in the state s' and evolves infinitely or finishes once at a state that has no outgoing transitions.

In our case, states of LTSs are typed graphs. The transitions are associated with graph transformation rules and are labelled with the graph transformation rule names.

Example 3.4.2 (Graph transformation rule system for behavioural semantics of Petri nets)**.** *We define the behavioural semantics of Petri nets by means of graph transformations. Since we have already the run-time graph T^{rt} (see Example 3.4.1), we need to specify the behaviour of tokens with graph transformation rules over T^{rt}. Note that the graph transformation rules must not change the syntactic structure of a graph, but only move a* Token*-node. The first rule we specify is a* pnInitial *rule (Figure 3.22), which creates* Token*-nodes for all* Place*-nodes that are marked as initial. The* pnMoveToken *rule (see Figure 3.23) allows a* Transition*-node to* fire Token*-nodes. If a* Transition*-node fires, one token is consumed from each input* Place*-node and moved to all the output* Place*-nodes. If there is at least one* Place*-node that does not connected with a* Token*-node then this rule cannot match the* Transition*-node and therefore the* Transition*-node cannot fire.*

We define a rule system as a partial mapping $\mathcal{RS}_{PN} : \mathsf{Sym} \rightharpoonup \mathsf{Rule}$, where Rule *is a set of graph transformation rules and* Sym *is a universe of rule names. We let $dom(\mathcal{RS}_{PN}) = \{\mathsf{pnInitial}, \mathsf{pnMoveToken}\}$ be the names of semantic rules in the rule system for the Petri nets (see Figures 3.22 and 3.23).*

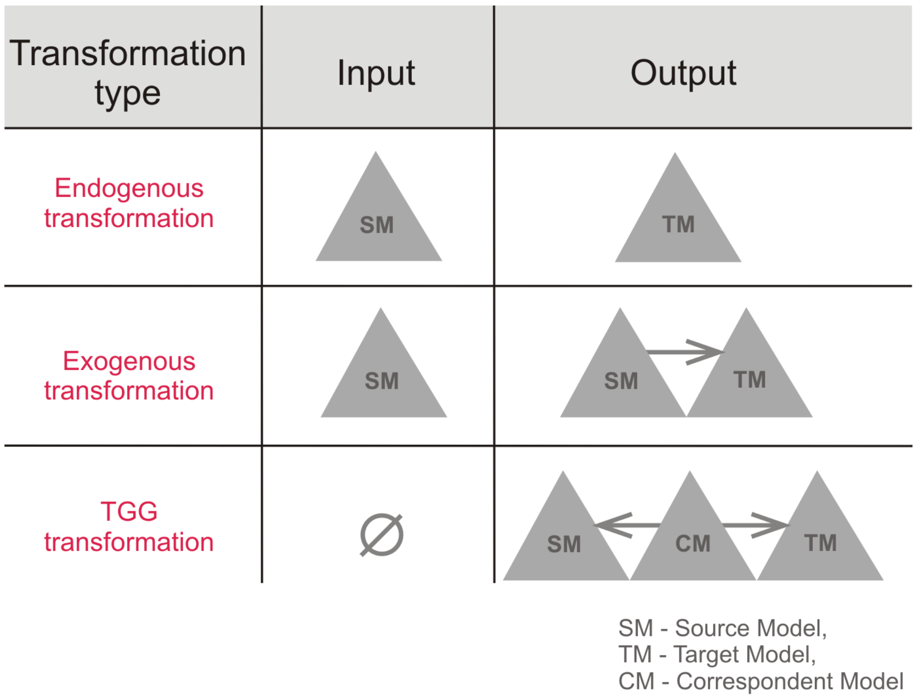


Figure 3.24: Three types of model transformations

3.5 Model Transformation Based on Graph Transformations

Since we consider modelling languages as graph languages, a model transformation could be specified by means of graph transformations.

We distinguish three types of graph transformations (see Figure 3.24). *Endogenous* transformations destroy the source graph while building the target graph. *Exogenous* transformations build a target graph by keeping a source graph, the result of such transformations is a graph, which contains both source and target graphs. The *Triple Graph Grammar (TGG)* transformations [K05̈, GK10, GK07] build two graphs simultaneously. The TGG transformations take an empty graph as an input and generate a source and a target graph as output.

The main idea of the TGG technique is that graphs are separated into three subgraphs, each being typed over its own type graph (see Figure 3.25). Two of these subgraphs evolve simultaneously while the third keeps correspondences between them. This correspondence node represents the third graph in TGG. As opposed to traditional transformation where the source model is given and then the source model is replaced by the target model, TGG transformations build the models simultaneously, matching each part of the source model to the target one. This allows to keep correspondences between transformed elements and to prove certain properties of the corresponding graphs.

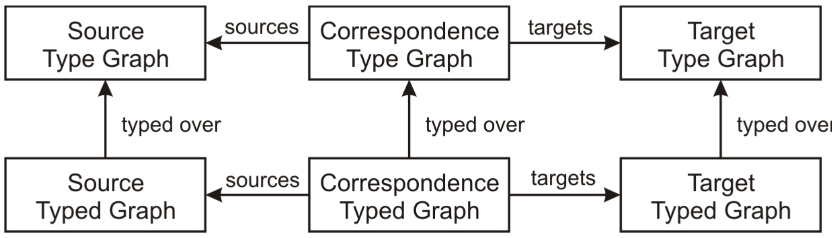


Figure 3.25: Graphs used in TGG technique

In the following, we formally define a model transformation based on the TGG technique. Let TGG rules build a graph typed over a type graph, that is a tuple $T^{\mathsf{st}} = T^{\mathsf{st}}_{SM} \times T_{CM} \times T^{\mathsf{st}}_{TM}$, where T^{st}_{SM} and T^{st}_{TM} are the type graphs that correspond to a source model and a target model respectively. Let $\mathcal{G}^{\mathsf{st}}$ denote a set of graphs obtained by applying the TGG rules on an empty start graph and a graph $G \in \mathcal{G}^{\mathsf{st}}$ is an output graph of the TGG rules. Note that graph G contains a source graph G_{ST} and a target graph G_{TM}. To obtain the actual translation, we restrict $\mathcal{G}^{\mathsf{st}}$ to the type graphs T^{st}_{SM} and T^{st}_{TM}. Using Definition 5 of type graph restriction, the model transformation MT is defined as follows:

Definition 13 (Model transformations)**.** Let G_{SM} be a source graph and G_{TM} be a target graph. We call $MT(G_{SM}, G_{TM})$ a *model transformation*, if it generates a graph $G \in \mathcal{G}^{\mathsf{st}}$ such that $G_{SM} = \pi_{T^{\mathsf{st}}_{SM}}(G)$ and $G_{TM} = \pi_{T^{\mathsf{st}}_{TM}}(G)$.

Example 3.5.1 (Example of a model transformation by means of TGG)**.** *We use a small example from the refactoring area to demonstrate a model transformation specified by means of TGG rules. The TGG technique does not suit well the refactoring area, however this example emphasizes the difference of TGG technique to the traditional transformation. We transform Petri nets into coloured Petri nets [JK09, Jen97]. We use a very simplified definition of coloured Petri nets: they are defined similar to Petri nets, except each place in coloured Petri net has an inscription, which determines the set of token colours that the tokens in that place are allowed to have. The set of possible token colours is specified by means of a integer type.*

We specify a type graph for coloured Petri nets by using the type graph of Petri nets. We add the prefix C to the names of nodes, since the type graph of Petri nets and the type graph of coloured Petri nets must be combined in one graph and the names of nodes must be unique. Places in coloured Petri nets have an additional value attribute of type integer. The correspondence graph consists of two nodes of

CN*1 and* CN*2 types which specify the correspondences between the source and the target model. See the result in Figure 3.26.*

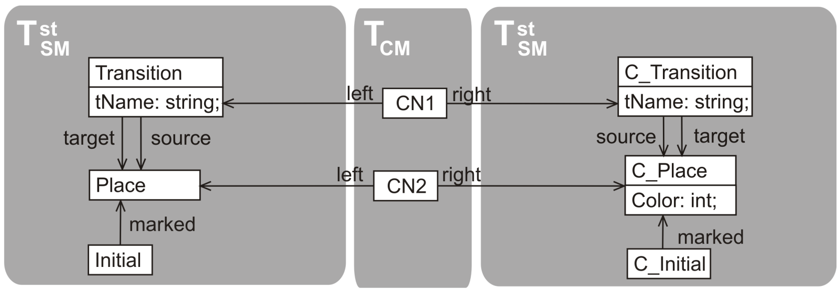


Figure 3.26: Type graph for the TGG transformations

We present one sample TGG rule defined over $T^{st}_{SM} \times T_{CM} \times T^{st}_{TM}$, which creates a Place*-node marked as initial and a corresponding structure of coloured Petri nets (Figure 3.27).*

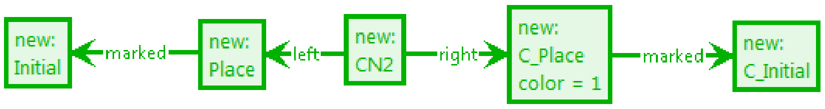


Figure 3.27: Sample TGG rule

3.6 Graph Transformation Tool

There is a number of tools to define model transformation and behavioural semantics by means of graph transformations (e.g. AGG [EEPT06b], ATL [JABK08], TGG [KÖ5] tools). Our main criteria for the tool are the following:

- Formal specification of graph transformations.
- Specification of graph transformation rules in terms of SPO approach.
- Support for attributed graphs.
- Possibility to implement TGG transformations.
- Examples of implemented language semantics.

We chose the Groove tool [Ren04a] for specification of our behaviour semantics and model transformation, since the tool satisfies our criteria. In follow, we detail our reasons for this choice.

Groove is a powerful tool, because despite the basic graph transformation features like creation and deletion of nodes and edges, Groove support some more

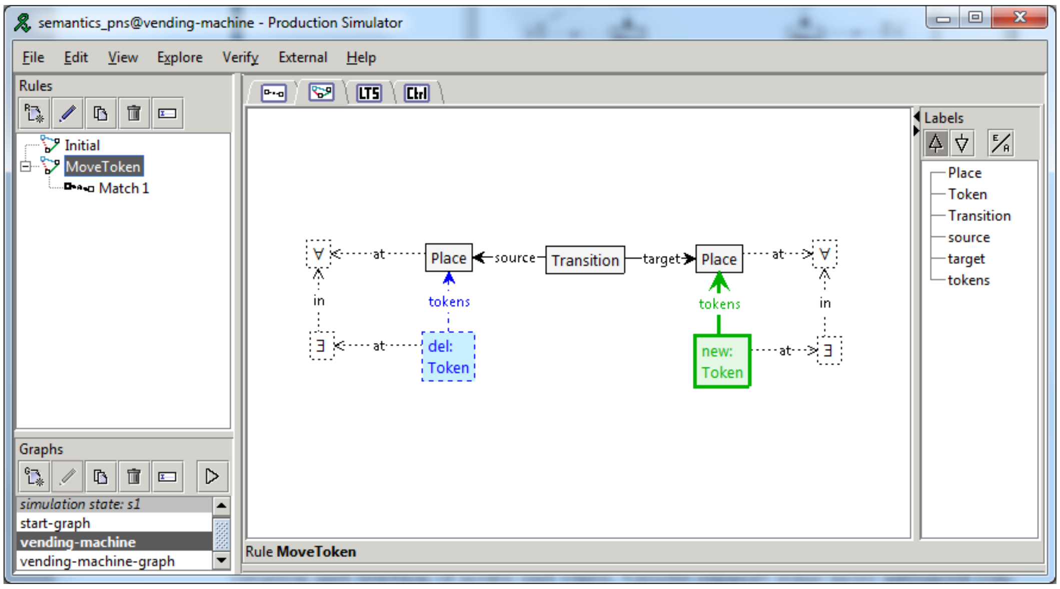


Figure 3.28: Screenshot of the tool Groove

advanced concepts. First, Groove supports attributed graphs. Secondly, a powerful notion of UQS is implemented in Groove. Third, it is possible to define graph transformations in TGG style as well.

Additionally, we had already an experience with the Groove tool [EKR+08], when we used the behavioural semantics of UML Activity diagrams and TAAL language, which are implemented in Groove, in order to implement a model transformation between these languages. Since we had a successful experience, it was an obvious choice to use this tool for us further.

Figure 3.28 shows a screenshot of Groove. In the left side, the names of the transformation rules can be seen. The big compartment in the right shows the graph transformation rule for moving the Token-node in Petri nets. in the right column, there is a list of used labels (from Sym_V and Sym_E). In the bottom left corner, there is a list of possible start graphs.

3.7 Summary

This chapter provides the notions of graph and graph transformations, which we are going to use throughout the remainder of this thesis. It was also explained how graphs could be used for the definition of an abstract syntax. The graph transformations by turn could be used for the definition of behavioural semantics and a model transformation specification as well.

CHAPTER 4

Equivalence Relation on LTS

The notion of Labelled Transition System (LTS) was already introduced in Chapter 3 as a model that describes behaviour of a system. In this chapter, we are interested in an equivalence relation over LTSs, because it is closely connected with our original goal: to show the behavioural properties preservation during model transformation. Behavioural properties are specified over LTSs by means of a formal language, as for example CTL* (a superset of linear temporal logic and Computational Temporal Logic [CGP99, BC87, GM93]). A single property, such as "a program must always terminate", could be written as a CTL* formula $AF(FinalState)$, where AF stands for "always in the future" and $FinalState$ is the name of a graph transformation rule, which performs a step in an LTS that leads to a final state. If and only if formulas specified over CTL* hold for two LTSs, then these LTSs are called *CTL* equivalent* [Hol95]. Then, if we establish a connection between CTL* equivalence and equivalence relation over LTSs, we can achieve our original goal by comparison of two LTSs (see Figure 4.1).

4.1 General Approach

In this section we want to answer three questions, which we present and discuss further.

▷ *Question 1: How to specify a behavioural property formally?*

We want to specify behavioural properties over LTS with a formal language. However, we must take into account that we deal with modelling languages, which are

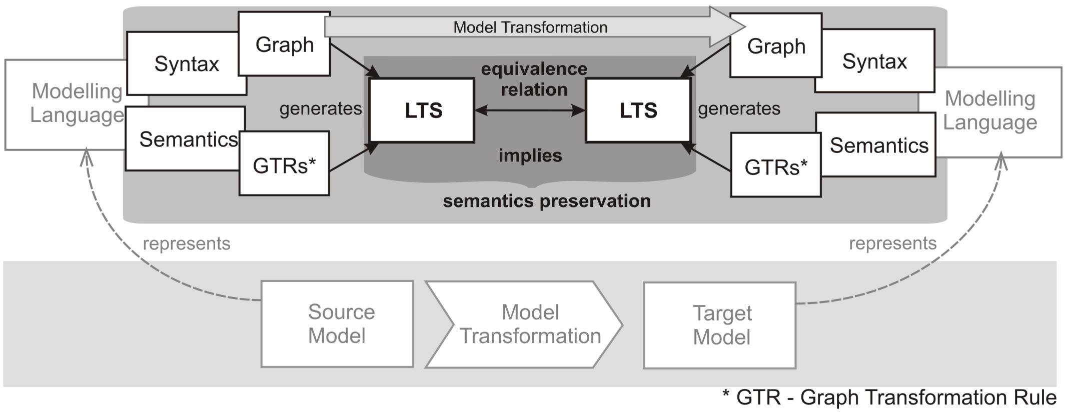


Figure 4.1: Discussion topic of this chapter

specified by means of graph transformations. It means that models are graphs and semantics of the models is described by a graph transformation system, which is applied to a certain graph. Each applied graph transformation rule, also called "semantic rule", is an *action*. Then, an LTS for a graph is a set of all possible action sequences. Thus, for each action there is a labelled transition in the generated LTS, where each label corresponds to a graph transformation rule name.

There are two choices to specify behavioural properties over an LTS with due regard for its peculiarities. It is possible to specify properties either over the states or over the transitions of LTS. Since the states in our LTS are graphs – a complicated structure, which includes typing and references, – a state-based logic over our LTS is harder to specify, than a transition-based or action-based logic, which is based on logic over the labels of transitions that are the names of semantic rules. Therefore, we use an action-based version of the branching time logic CTL*, which is called ACTL [NFGR93]. According to [NV90], ACTL has the same expressive power as CTL*.

▷ *Question 2: How to specify an equivalence relation over LTSs?*

A model transformation is defined over the syntax of modelling languages. We want to check whether the defined model transformation preserves a behavioural semantics. Therefore, we compare the behaviour of two modelling languages. For this, we define a mapping function over the names of graph transformation rules (Figure 4.2). It could be possible that some graph transformation rules perform a behaviour that could not be translated into another language. Such graph transformation rules stay unmapped and later perform invisible actions in LTS. Thus, we compare a part of behaviour, which can be translated into another language.

Owing to the mapping function, which maps some graph transformation rule to invisible actions, we can specify two major problems that arise during the comparison of behavioural models (in our case LTSs) of different languages. The first

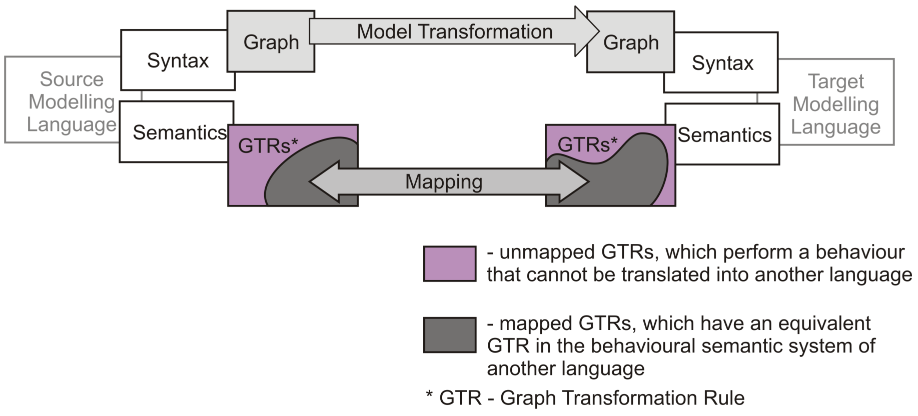

Model Transformation
Graph
Syntax
Source Modelling Language
Semantics
GTRs*
Mapping
Graph
Syntax
Target Modelling Language
Semantics
GTRs*
- unmapped GTRs, which perform a behaviour that cannot be translated into another language
- mapped GTRs, which have an equivalent GTR in the behavioural semantic system of another language
* GTR - Graph Transformation Rule

Figure 4.2: Comparison of behavioural models is based on the mapping function defined over the names of graph transformation rules

problem relates to disjoint label sets of LTSs. Transitions in LTS are labelled with the names of behavioural semantic rules (or graph transformation rules). In order to compare two LTSs one needs to match the transitions of these LTSs, which perform the similar behaviour. Since different languages are defined by completely different graph transformation systems, the LTSs of different languages have different labels. Secondly, part of behaviour could not be translated into another modelling language. Then, a variety of transitions could not be matched.

The problems described above could be solved by establishing a mapping function over the label sets of both LTSs in a similar way that a mapping over the names of graph transformation rules is defined. It means that labels of transitions, which correspond to actions that describe similar behaviour, are mapped to a common label. Actions, which could not be translated into another language, become *invisible* and their correspondent labels are mapped to the label called "tau". Thus, we compare not original LTSs, but their corresponding mapped structures.

In general we perform an abstraction of original LTSs that is based on definition of a label set, which both LTSs are mapped to. Figure 4.3 explains the principle of the abstraction. $Q_1 = Q(G_1)$ is an LTS generated for a graph G_1, which is from source language. $Q_2 = Q(G_2)$ is an LTS generated for a graph G_2, which is a result of a model transformation between source and target modelling languages. Q_1 and Q_2 are defined over the label sets L_1 and L_2, respectively. The label sets L_1 and L_2 are both mapped to a common set $L = \tau \cup Visible$, which consists of an element τ and a set labels that correspond to some visible action. Then, the LTSs Q'_1 and Q'_2, which are the result of the mapping, can be compared, i.e. the equivalence relation could be established.

$\triangleright$ *Question 3: How to prove that equivalence relation implies ACTL equivalence?*

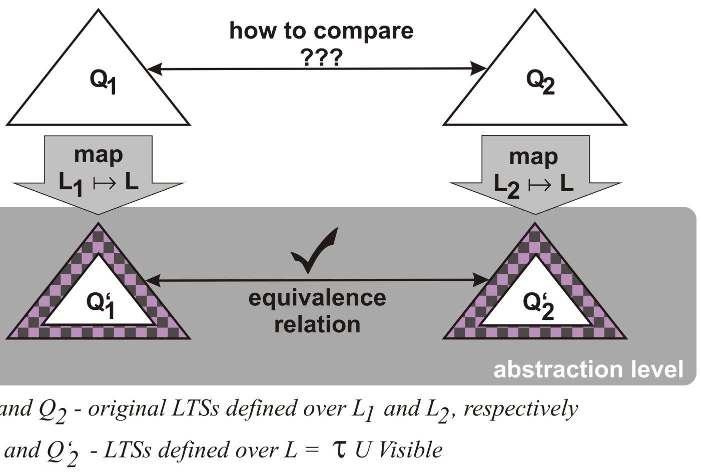

how to compare ???
Q1
Q2
map L1 ↦ L
map L2 ↦ L
Q'1
Q'2
equivalence relation
abstraction level
Q1 and Q2 - original LTSs defined over L1 and L2, respectively
Q'1 and Q'2 - LTSs defined over L = τ U Visible

Figure 4.3: Comparison of LTSs on the level of abstraction

The notion of *bisimulation equivalence* (also called *observational equivalence*) is one of the best known notions of equivalences over LTSs [PdRV95]. Intuitively, two systems are *bisimilar*, if they can perform the same sequences of actions to reach bisimulation equivalent states. Bisimulation equivalence is called *strong*, when all labels of LTS are considered visible (i.e. no labels are hidden from observation), and *weak*, when some actions are ignored or considered to be internal and thus invisible. It is easier to establish a weak bisimulation relation, than e.g. strong, on LTSs of two different languages, because there could be some transitions which have no equivalent transitions in another language.

A bisimulation equivalence relation is closely connected with ACTL equivalence. There is some work done in the direction of implementation of the connection between bisimulation and CTL* [BCG88]. That is a very strong hint for the solution how to establish a connection between weak bisimulation and ACTL equivalence. Then, a weak bisimilarity of two LTSs implies our original goal, i.e. behavioural properties preservation during model transformation:

$$Q(G_1) \models \varphi \quad \Rightarrow \quad Q(G_2) \models \chi(\varphi)$$

Here, a behavioural property φ is specified with the ACTL in the language L_1. χ is an interpretation of φ for the language L_2. Due to the fact that we compare LTSs on the level of abstraction (Figure 4.3), the LTSs are labelled over the same set. Therefore, there is no need for interpretation of a property φ on the abstraction level. The goal of this chapter to show how to prove the following statement for *any* behavioural property φ:

$$Q'(G_1) \models \varphi \quad \Rightarrow \quad Q'(G_2) \models \varphi \tag{4.1}$$

The interpretation of properties (i.e. the meaning of function χ) is explained in the next chapter in detail.

We proceed as follows. This chapter repeats the definition of LTS (Section 4.2), then we discuss the notion of weak bisimulation equivalence between two LTSs (Section 4.3). We introduce ACTL over states of an LTS in Section 4.4. In particular we explain how to specify a property for a graph transformation language by means of ACTL. Finally, we show that weak bisimulation equivalence implies ACTL equivalence in Section 4.5.

4.2 Transition Systems

In this section, we repeat the definition of LTS, which was introduced in Chapter 3 as a structure that describes the behaviour of a model. An LTS is basically a directed graph, where nodes represent *states* and edges model *transitions.* A state describes the system at a certain execution state of graph transformation system. A transition represents an action being performed to change a state of the system. Transitions are labelled over the set of graph transformation rule names. A transition could be marked as invisible, denoted as τ. This means that this transition corresponds to an invisible action.

Definition 14 (Labelled Transition System)**.** An *L-labelled transition system* (LTS) is a structure $Q = \langle S, \rightarrow, \iota, L\rangle$, where S is a set of states and $\rightarrow \subseteq S \times L \times S$ is a set of transitions labelled over some set of labels L. Furthermore $\iota \in S$ is the start state.

Notes:
- LTS is called *finite*, if S is finite.
- An invisible action τ is assumed to be in L.
- For convenience, we write $s \xrightarrow{\alpha} s'$ to denote a transition in LTS, or (s, α, s').
- For a given LTS Q, we use S_Q, $\rightarrow_Q$, etc. to denote its components.

Example 4.2.1 (Example of an LTS)**.** *We consider a simple example of a Petri net model (the Petri nets language was introduced in a previous chapter). The Petri net in Figure 4.4 (left) models a preliminary design of a beverage vending machine. The machine can except payment and deliver water. The Petri net transition with an attribute* pay *denotes the insertion of coins, while the transition* d_water *denotes delivery of a drink.*

The behaviour of the Petri nets is defined by a token flow that means that a `Token` *flows through the Petri nets transitions and could be put only in places. When it passes a transition a single action occurs. Such behaviour is described with a graph transformation system, which consists of two graph transformation rules:* pnMoveToken *and* pnInitial*, which are depicted in Figure 4.5. An LTS generated for a*

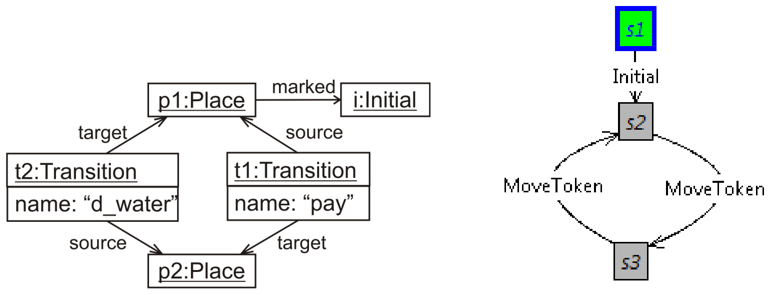


Figure 4.4: Example of a Petri net (in the left) and a corresponding LTS (in the right)

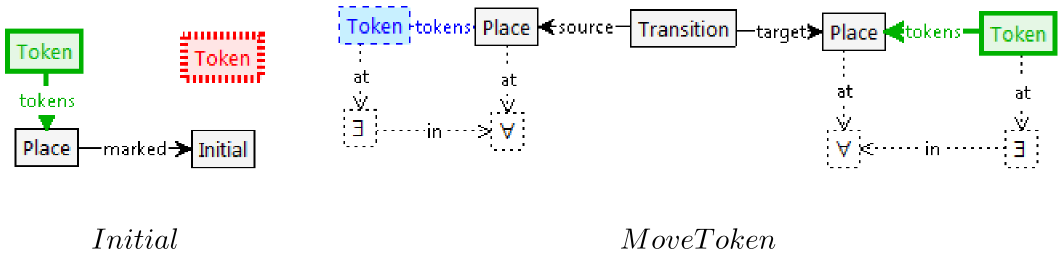


Figure 4.5: Graph transformation rules *pnInitial* and *pnMoveToken* from a graph transformation system $\mathcal{RS}_{PN}$

graph, which models the beverage vending machine described above, is presented in Figure 4.4 (in the right).

*The states of the LTS are represented as squares with names depicted inside. The transitions of the LTS are the labelled edges, which correspond to actions, i.e. graph transformation rules applied to the graph. The transition label represents the name of a graph transformation rule (*pnMoveToken *or* pnInitial*). The initial state is indicated by having a bold border and denoted as* $s1$*. The state space of our LTS consists of* $S = \{s_1, s_2, s_3\}$*. The initial state is* $\iota = s_1$*. An example transition is:*

$$s_2 \xrightarrow{pnMoveToken} s_3$$

4.3 Bisimulation as Type of Behavioral Equivalence

Consider two LTSs Q_1 and Q_2 with different sets of labels L_1 and L_2, respectively. To compare these LTSs we define a suitable mapping of the sets L_1 and L_2 to a common set L. This mapping defines labels, which correspond to the graph transformation rules that perform similar behaviour. There are could be also labels that have no correspondences, such labels correspond to invisible actions (i.e. transitions hidden from observation or not translated into another language) and are denoted as τ. Thus, we do not compare the original LTSs, but rather their corresponding mapped structures.

Further, we define a mapping function on sets of labels. The set of labels consists of labels of two types: (1) labels which correspond to a graph transformation rule that performs an invisible aciton, (2) labels which correspond to a graph transformation rule that performs a visible aciton. The mapping function defined below maps labels of the first type to τ and labels of the second type to an element different to τ.

Definition 15 (Function for visible and invisible labels)**.** Let $\alpha \in L_Q$ be a label of transition in Q, then the function $\widehat{\ }: L_Q \to L$ is as follows:

- $\widehat{\alpha} = \tau$, if α is a label of transition, which corresponds to an invisible action,
- $\widehat{\alpha} = \alpha$, if α is a label of transition, which corresponds to a visible action.

For states $s, s' \in S$ and a label α, to denote a sequence of transitions $s \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} s'$, we write $s \overset{\alpha}{\Longrightarrow} s'$. We also use $\overset{\varepsilon}{\Longrightarrow}$, which stands for $\xrightarrow{\tau}{}^{*}$.

Bisimulation equivalence aims to identify LTSs with the same branching structure, and which thus can simulate or mimic each other in a stepwise manner. Roughly, a transition system Q_1 can simulate transition system Q_2 if every step of Q_1 can be matched with one or more steps in Q_2. We firstly introduce the weak bisimulation relation as a binary relation over the set of states. Then we define two weak bisimilar LTSs. Bisimulation is defined as the largest relation satisfying certain properties.

Definition 16 (Weak bisimulation)**.** Let $\alpha \in L_{Q_1}$ and $\beta \in L_{Q_2}$ be transition labels in the LTSs Q_1 and Q_2, respectively. *Weak bisimulation* is a relation over the set of states, i.e. $\approx \subseteq S_1 \times S_2$, here $S_1 = S_{Q_1}$ and $S_2 = S_{Q_2}$, such that $\forall s_1 \in S_1$ and $\forall s_2 \in S_2$ whenever $s_1 \approx s_2$ means that

- If $s_1 \xrightarrow{\alpha} s_1'$, then $\exists s_2' \in S_2$ with $s_2 \overset{\widehat{\alpha}}{\Longrightarrow} s_2'$ such that $s_1' \approx s_2'$;
- If $s_2 \xrightarrow{\beta} s_2'$, then $\exists s_1' \in S_1$ with $s_1 \overset{\widehat{\beta}}{\Longrightarrow} s_1'$ such that $s_1' \approx s_2'$.

States s_1 and s_2 are called *weak bisimilar states* if $s_1 \approx s_2$.

It means that every outgoing transition of s_1 must be matched with an outgoing sequence of transition of s_2, and vice versa. Figure 4.6 summarises these two conditions.

$$\begin{tikzcd}
s_1 \arrow[d, "\alpha"'] & \approx & s_2 & & s_1 \arrow[d, "\alpha"'] & \approx & s_2 \arrow[d, Rightarrow, "\widehat{\alpha}"'] \\
s_1' & & & \text{can be complemented to} & s_1' & \approx & s_2'
\end{tikzcd}$$

$$\begin{tikzcd}
s_1 & \approx & s_2 \arrow[d, "\beta"'] & & s_1 \arrow[d, Rightarrow, "\widehat{\beta}"'] & \approx & s_2 \arrow[d, "\beta"'] \\
 & & s_2' & \text{can be complemented to} & s_1' & \approx & s_2'
\end{tikzcd}$$

Figure 4.6: Conditions of weak bisimulation (reproduced from [BK08])

Note that we consider that for all labels $\alpha \in L_{Q_1}$ either $\widehat{\alpha} = \tau$ or there exists $\beta \in L_{Q_2}$ such that $\widehat{\alpha} = \widehat{\beta}$. The riverse is stated for all labels $\beta \in L_{Q_2}$, too.

Definition 17 (Weak bisimilar LTSs)**.** Two labelled transition systems Q_1 and Q_2 are *weak bisimilar*, iff there exists a weak bisimularity $\approx$ such that

(1) $\iota_1 \approx \iota_2$
(2) $\forall s_1 \in S_1$, $\exists s_2 \in S_2$ such that $s_1 \approx s_2$,
(3) $\forall s_2 \in S_2$, $\exists s_1 \in S_1$ such that $s_1 \approx s_2$.

Condition (1) asserts that every initial state of Q_1 is related to an initial state of Q_2, and vice versa. Condition (2) states that every state from Q_1 must be in a weak bisimulation with at least one state from Q_2; the reverse is stated by (3).

The diagram a), b), c) shown in Figure 4.7 summarizes the structure of weak bisimilar states. These diagrams denote instances of weak bisimilar LTSs. We have used the triangles to indicate weak bisimilar LTSs (i.e. $Q_0 \approx Q_1 \approx Q_2 \approx Q_3$) that include a state placed on top of the triangle. In the diagram a) there are two states s_0 and s_1 that are in a weak bisimulation with a state s_2, because all states reachable from s_0 are also reachable from s_1. In diagram c) the states s_3 and s_4 are also weak bisimilar with the state s_2 (transition labelled as τ steps are not considered, since they do not break the conditions for weak bisimilarity).

Example 4.3.1 (Not bisimilar LTSs)**.** *Let consider three LTSs, see Figure 4.8, depicting Q_1 in the left, Q_2 in the middle and Q_3 in the right. It follows that Q_1 and Q_2 are not weak bisimilar, since the state s_1 in Q_1 can not be simulated by a state in Q_2. This can be seen as follows. The only candidate for simulating state s_1 is the state s_4. However, s_1 can not simulate all transitions of s_4 in Q_2: the possibility for b is missing. Thus, $Q_1 \not\approx Q_2$. Q_1 and Q_3 are not weak bisimilar due to the same reason.*

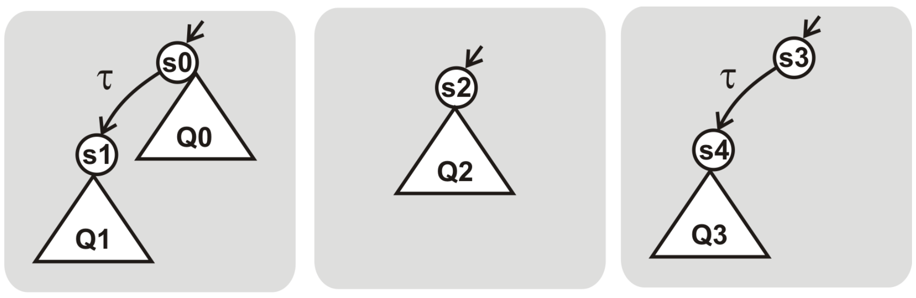


Figure 4.7: Diagrams a), b), c) denote three LTSs that are mutually weak bisimilar

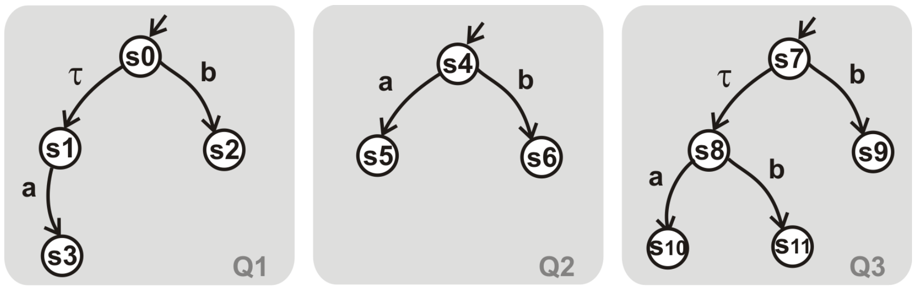


Figure 4.8: An example of labelled transition systems, here Q_1 and Q_2 are NOT weak bisimilar, Q_2 and Q_3 are weak bisimilar

Please note that the LTSs Q_2 and Q_3 are weak bisimilar, because the state s_8 in Q_3 can be simulated by the state s_4 in Q_2. All the other states are obviously in a weak bisimulation with at least on state from Q_2. The reverse statement for the states of Q_2 holds, too.

4.4 Properties Specification over LTS

In this section, we explain how to specify behavioural properties over a modelling language. For this we use a behavioural model, i.e. an LTS, and a formal language that could specify a wide range of properties over an LTS by means of formulas. Since we deal with special LTS, where states are graphs and transitions correspond to graph transformation rules, we searched for a formal language adjusted for the specification of properties for our LTS. We explain our choice, which is an action-based logic, called ACTL. Further in this section, we define the syntax of ACTL. After we provide a satisfaction relation for ACTL formulas, we explain how to specify a property with ACTL over a language, defined by means of graph transformation rules, and how to verify this property against an LTS.

4.4.1 Why ACTL

We deal with special LTSs, where states are graphs and transitions are labelled with names of graph transformation rules (see Section 3.4). Therefore, we can either specify behavioural propertied over the states or over the transitions. For this, there are exists two types of formal languages, which allow to specify formulas over LTSs: action-based and state-based. Action-based logic specifies formulas over the transition labels of LTSs and state-based logic specifies formulas over the state labels. Graph, which is a state in our LTS, is not a suitable construction to be a label, since it consists of complicated types. Therefore, we use transition labels to specify the behavioural properties over LTS.

Since we are interested in an action-based logic, we have a choice between two well-known logics: Hennessy-Milner Logic (HML) and active-based version of CTL*, called ACTL. We are interested in a more expressive one, because it allows to specify a wide range of behavioural properties. It is also important the existence of model checkers, which allow to verify the formulas.

In compliance with our demands ACTL has more advantages to HML. We found that, firstly, there is a bigger alternative of model checkers to verify CTL* formulas (state-based version of ACTL) than HML formulas, such as EMC [CES86], or a CTL model checker for graph transformation systems implemented in Groove [KR06] (there, an LTS is transformed into a Kripke structure, then the properties are specified by CTL* over a Kripke structure). Secondly, in some approaches it is mentioned that HML is not expressive enough:

> *" ... we have logics, like Hennessy-Milner logic, that are not sufficiently expressive ..., that require using fixed pointers and lack directness in describing systems properties (see e.g. [Lar88])."*[NFGR93]

Thereby, our choice was the ACTL logic.

4.4.2 Syntax of ACTL

This subsection describes the syntactic rules according to which formulas in ACTL can be constructed. Before we start to explain the syntax, we introduce the relevant notion that have been used to interpret ACTL formulas on an LTS.

Definition 18 (Notation for LTS)**.** Let $Q = \langle S, \rightarrow, \iota, L\rangle$ be an LTS, then
− A nonempty (finite or infinite) sequence $\pi = (s_0, \alpha_0, s_1)(s_1, \alpha_1, s_1)\ldots$, where (s_i, α_i, s_{i+1}) (a transition in LTS) with $i \geq 0$, is called a *path* from s_0. π^i will denote the *suffix* of π starting at s_i. We write $L(\pi^i)$ to denote a label α_i of transition (s_i, α_i, s_{i+1}).
− If a path is infinite or it can not be extended anymore because it ends in a state without outgoing transitions, it is called a *fullpath.*

– A *run* from $s \in S$ is a pair $\rho = (s, \pi)$, where π is a path from s; we write *first(ρ)*$= s$ and *path(ρ)*$= \pi$.
– A *maximal run* is a run whose second element is a fullpath. We write *maxrun(s)* for the set of maximal runs from s.
– We let π range over paths and $\rho, \sigma, \ldots$ over runs.

Definition 19 (ACTL formula)**.** Let **AP** be the set of atomic proposition names. A formula is either:

- a if $a \in \mathbf{AP}$;
- if f and g are formulas, then $\neg f$ and $f \wedge g$ are formulas;
- f is a formula, then $\exists f$ is a formula;
- if f and g are formulas, then $f\ U\ g$ ("until") is a formula;
- if f is a formula, then Xf ("nexttime") is a formula.

Moreover, we specify additional operators:

- $true$ states for $\neg(a \wedge \neg a)$, where $a \in \mathbf{AP}$,
- $false$ states for $\neg true$,
- $f \vee g$ states for $\neg(\neg f \wedge \neg g)$,
- $\forall f$ states for $\neg\exists\neg f$,
- Ff states for $true\ U\ f$,
- Gf states for $\neg F \neg f$.

4.4.3 Semantics of ACTL

ACTL formulas are interpreted over runs, which are defined as a pair of a state and a path of an LTS. Formally, given an LTS Q, the semantics of ACTL formulas is defined by a satisfaction relation (denoted by $\models$). For the formulas, $\models$ is a relation between a run with a maximal path fragment in Q and formulas. We write $\rho \models f$. The intended interpretation is: $\rho \models f$ if and only if run ρ satisfies the formula f.

Definition 20 (Satisfaction relation for ACTL)**.** Let $a \in \mathbf{AP}$ be an atomic proposition, $Q = \langle S, \rightarrow, \iota, L\rangle$ be an LTS, f and g be ACTL formulas. The satisfaction relation by a run $\rho = (s_1, \pi)$ is defined inductively by:

- $\rho \models a$ iff $\exists k \geq 1$ such that $\forall i : 1 \leq i < k$ (s_i, τ, s_{i+1}) (transitions are labelled as "tau") and $a \in L(\pi^k)$,
- $\rho \models \neg f$ iff not $\rho \models f$,
- $\rho \models f \wedge g$ iff $\rho \models f$ and $\rho \models g$,
- $\rho \models \exists f$ iff there exists a run $\theta \in$ *maxrun(first(ρ))* such that $\theta \models f$,
- $\rho \models f\ U\ g$ iff $\exists k \geq 1$ such that $(s_k, \pi) \models g$ and $\forall i : 1 \leq i < k$ $(s_i, \pi) \models f$,
- $\rho \models Xf$ iff $\exists k \geq 2$ such that $(s_k, \pi) \models f$, $\forall i : 1 \leq i < k-1$ (s_i, τ, s_{i+1}) and $\tau \notin L(\pi^{k-1})$.

The LTS Q satisfies ACTL formula φ if and only if φ holds in all runs from initial states of Q:

$$Q \models \varphi \text{ if and only if } \forall \rho : first(\rho) = \iota_Q \quad \rho \models \varphi$$

Thereby, the operators introduced before can be interpreted as follows:

- $Q \models \exists f$ - f is satisfied by *some* run $\rho = (s_1, \pi)$,
- $Q \models \forall f$ - f is satisfied by *all* runs $\rho = (s_1, \pi)$,
- $Q \models Ff$ - f is satisfied by *some* runs *sometimes* in the "future",
- $Q \models Gf$ - f is satisfied by *all* runs from now and forever or *globally*,
- $Q \models f\ U\ g$ - there exists a run ρ, where there is a state along a path s_1, g is satisfied by some run $\rho_1 = (s_1, \pi)$, and for all states s_i prior to the state s_1 (in the run ρ) f is satisfied by all runs $\rho_2 = (s_i, \pi^i)$,
- $Q \models Xf$ - there exists a run ρ, where there is a state along a path s_k, f is satisfied by some run $\rho_1 = (s_k, \pi)$, for all states s_i prior to the state s_{k-1} the transitions are labelled as τ, i.e. $\tau \in L(\pi^i)$, and $\tau \notin L(\pi^{k-1})$.

4.4.4 Behavioural Properties Specification with ACTL

In this subsection, we explain how to specify a behavioural property with an ACTL formula. An explanation is supported with an example of a behavioural property and demonstration how the formula could be verified. Prior to this, we briefly review the process of behavioural semantics specification for graph languages.

Elements of a graph language are graphs G. Behavioural semantics of a graph language is specified by a rule system $\mathcal{RS}$, which is a set of graph transformation rules defined over a graph G (see Chapter 3). A change of a graph G is specified by a graph transformation rule, which consists of three patterns: $p = \langle \mathcal{N}_p, L_p, R_p \rangle$, where $\mathcal{N}_p$ is a negative application condition, the set $L_p \cap R_p$ specifies a match for p, the sets $L_p \backslash R_p$ and $R_p \backslash L_p$ describe elements to be deleted and to be created, respectively. The rule p is applicable to a graph G_0, if L_p is matched to some subgraph of G_0 such that there is no pattern $\mathcal{N}_p$ in this match. The rule p turns the graph G_0 into graph G_1, be replacing a matched subgraph of G_0 with a pattern R_p. A graph transformation system generates an LTS, where transitions are labelled with the names of graph transformation rules from $\mathcal{RS}$, e.g. an example of a transition in an LTS is $G_0 \xrightarrow{p} G_1$.

Now we want to use a graph transformation rule not only as a structure, which specifies the changes on graphs, but also as a structure that constitutes structural properties. For this, we consider a left-hand side pattern L_p and a negative application condition $\mathcal{N}_p$ of a graph transformation rule p. They specify the conditions for a graph structure G_0. Similarly, the right-hand side pattern R_p specifies the conditions for a graph structure G_1. For example, if we consider the graph transformation rule $p = \mathsf{pnInitial}$ (see the example with a beverage machine in Section

4.2), the right-hand side pattern of this rule specifies that a Place-node marked as initial must be connected with a Token-node. It could be also interpreted as a behavioural property, which says that the system is activated. Thereof, the pnInitial rule could be considered as a structural property that describes a state during the semantics execution. Thus, a graph transformation rule is considered further not only an action, but *it also constitutes a structural property of a graph.* In the follow, we explain how to specify a required behavioural property with a help of a graph transformation rule.

The behavioural property is technically specified with an ACTL formula over a set of graph transformation rule names. It means that a graph transformation rule describes a state and the ACTL operators specify the conditions on this state. Let us consider again the example of a beverage vending machine from Section 4.2. We want to ensure that the dynamic element called Token is always created. For this, we use the pnInitial rule as a structural property. With a help of the ACTL operators we specify that this property always holds in the future:

$$\forall F(pnInitial)$$

It can be also possible that a required property is not specified by any of the graph transformation rules from $\mathcal{RS}$, as for example, the statement that the vending machine always delivers a drink after a payment was done. It is even more likely that a graph transformation rule describes a flow of a node through some structure, where concrete labels, as for example attributes or the node names, are not specified and therefore not depicted in the defined LTS, except the names of graph transformation rules from $\mathcal{RS}$. In our example, if we consider a transition $s_1 \xrightarrow{pnMoveToken} s_2$, it is not clear to which Transition-node (concerns a Petri net model) the pnMoveToken rule was applied, i.e. to the Transition-node with the attribute *d_water* or the attribute *pay*.

The problem discussed above can be solved if we specify additional graph transformation rules as a structural property instead of an action. It means that the left-hand and right-hand sides of a graph transformation rule are identical ($L_p = R_p$). Such rules have no structural effect on any state, but then they are depicted in an LTS as a loop transition, i.e. a transition of type ($s_i \rightarrow s_i$), and do not affect the weak bisimulation.

Thereby, by specifying additional graph transformation rules we extend the original rule system $\mathcal{RS}$. After our extension it is denoted as $\mathcal{RS}^+$ and consists of graph transformation rules from $\mathcal{RS}$, which describe the behaviour of the system, and graph transformation rules, which do not delete and do not create any nodes in a graph.

After we specified all required structural properties, we specify behavioural properties over the transitions of an LTS by an ACTL formula. It means that the names of graph transformation rules are used as atomic propositions. The diagram in

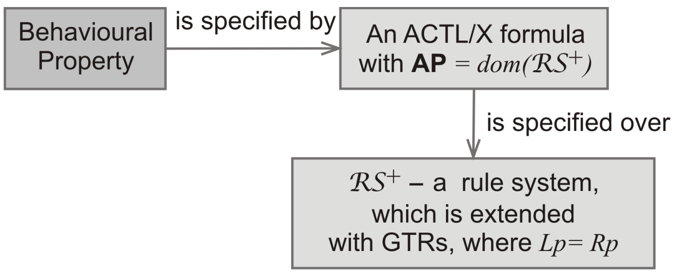

Behavioural Property
is specified by
An ACTL/X formula with **AP** = dom(RS+)
is specified over
RS+ – a rule system, which is extended with GTRs, where Lp= Rp

Figure 4.9: Diagram that shows how a behavioural property is specified for a graph language

Figure 4.9 summarizes the idea how a behavioural property is specified for a graph language.

Example 4.4.1 (Behavioural properties for a beverage vending machine)**.** *We illustrate two behavioural properties for a beverage vending machine, which accepts payments, provides a choice of a drink (water or juice) and delivers a chosen drink. We model this system with the Petri nets language. A model for the beverage vending machine is presented in Figure 4.11 in the left. We want to assure (1) that a system always delivers a drink after a payment was done, (2) that the system never delivers both drinks (water and juice).*

There are two graph transformation rules for the description of the Petri nets behavioural semantics (Initial and MoveToken from Figure 4.5), which do not specify the desirable properties. These graph transformation rules specify only flow of a Token*-node through a* Transition*-node without mentioning its attributes values. Thus, an attribute of a* Transition*-node is not depicted in a resulted LTS, that makes impossible to specify the required properties. However, we can extend a rule system* $\mathcal{RS}_{PN}$ *with graph transformation rules, which specify the desirable properties.*

We define three GTR, where L_p *and* R_p *are identical. Such graph transformation rules (see Figure 4.10) specify structural properties in the form of the rule. The graph transformation rule, denoted as pay, says that the payment is accomplished. The rule g_juice specifies that the system is ready to deliver a juice. The rule g_water says that the system is ready to deliver a water.*

Then, the properties (1) and (2) can be specified as:

$$\forall F(payU(g\_juice \vee g\_water)) \tag{4.2}$$

$$\forall G^{\neg}(g\_juice \wedge g\_water) \tag{4.3}$$

Property 4.2 is a liveness property specifying that specific actions, i.e. delivery of a water or a juice, must be reachable, after the payment was accomplished. Property

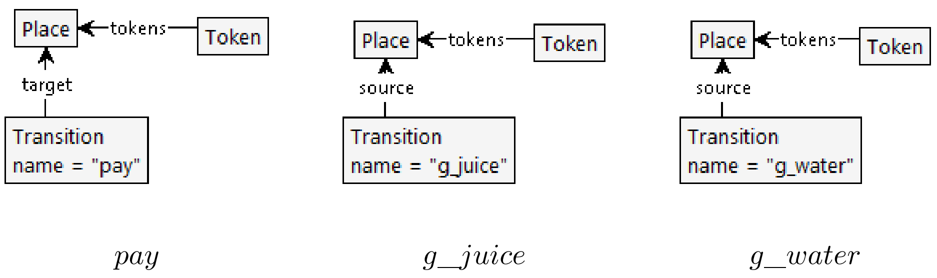


Figure 4.10: Graph transformation rules *pay*, *g_juice* and *g_water*, where graph structure is used not as a rule, but as a structural property

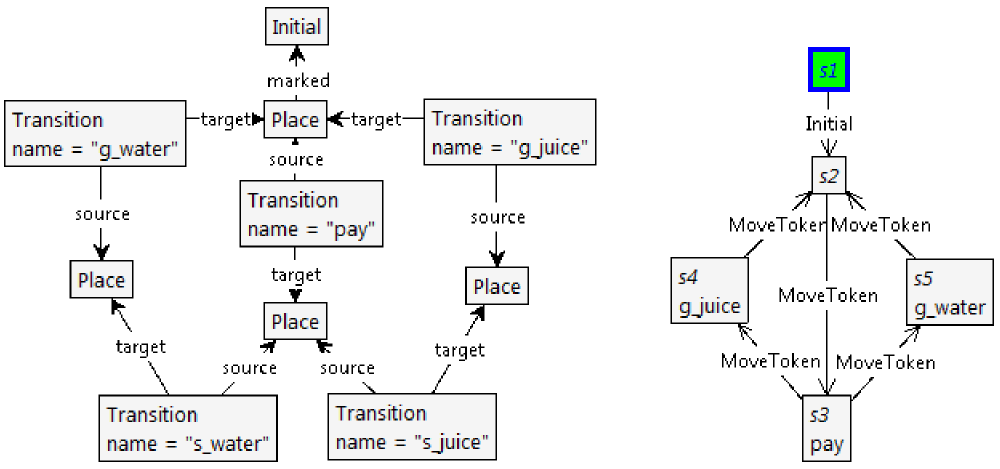


Figure 4.11: A graph which models the vending machine (in the left) and an LTS generated by an extended rule system (in the right)

4.3 is a safety property, specifying that the system may not contain a state, in which the delivery of both drinks is available.

We generate an LTS for the graph (from Figure 4.11 in the left) with the extended rule system, where $dom(\mathcal{RS}^+) = \{MoveToken,\ Initial, pay,\ g\_juice, g\_water\}$. *The result is depicted in Figure 4.11 in the right. The states* s_3, s_4, s_5 *have loop transitions, i.e. transition of type* (s_i, α, s_i), *for some* $i \geq 0$. *These transitions are produced by the extended rules (pay, g_juice, g_water), which do not change a graph.*

The formula 4.2 is valid for initial state since all paths starting in this place have *a direct successor state that satisfies pay and all paths, followed after the transition pay, contain either g_juice or g_water.*

The formula 4.3 holds in s_1, as there is no path that reaches a state, from which g_juice or g_water hold simultaneously.

4.5 ACTL Equivalence and Weak Bisimulation

This section considers the equivalence relation induced by the action-based logic ACTL, as well as performs a key theorem of this chapter about preservation of ACTL formulas by weak bisimulation.

Note that we consider the LTSs relabelled according to the function from Definition 15. Therefore, the LTSs are defined over the same label sets further in this section. The goal of this section is to show the connection of Equation 4.1 and weak bisimulation on the level of abstraction (see Figure 4.3).

4.5.1 ACTL Equivalence

Runs, which are defined as a pair of a state and a path in an LTS, are equivalent with respect to a logic whenever these runs cannot be distinguished by the truth value of any formulas of the logic. Stated differently, whenever there is a formula in the logic that holds in one run, but not in the other, these runs are not equivalent.

Definition 21 (ACTL equivalence)**.** Let Q_1 and Q_2 be LTSs over $\mathbf{AP}$

1. Runs ρ_1 and ρ_2 in Q_1 and Q_2, respectively, are ACTL equivalent, denoted $\rho_1 \sim \rho_2$ if

$$\rho_1 \models \varphi \text{ iff } \rho_2 \models \varphi \text{ for all ACTL formulas } \varphi \text{ over } \mathbf{AP}$$

2. Q_1 and Q_2 are ACTL equivalent, denoted $Q_1 \sim Q_2$, if

$$Q_1 \models \varphi \text{ iff } Q_2 \models \varphi \text{ for all ACTL formulas } \varphi \text{ over } \mathbf{AP}$$

Runs ρ_1 and ρ_2 are ACTL equivalent if there does not exist an ACTL formula that holds in ρ_1 and not in ρ_2, or, vice versa, holds in ρ_1, but not in ρ_2.

LTSs Q_1 and Q_2 are ACTL equivalent if there does not exist an ACTL formula that holds in ρ_1 such that $first(\rho_1) = \iota_{Q_1}$ and not in any ρ_2 such that $first(\rho_2) = \iota_{Q_2}$.

4.5.2 Preservation of ACTL Formulas by Weak Bisimulation

Before we present a key theorem of this chapter about preservation of ACTL formulas by weak bisimulation, let us firstly to define *weak bisimilar paths*, which we call also *corresponding paths.*

Paths π_1 and π_2 are corresponding paths if both paths can be divided into segments (separate subpaths)

$$s_{j_r,1}, s_{j_r+1,1}, s_{j_r+2,1}, \ldots, s_{j_{r+1},1},$$

and

$$s_{k_r,1}, s_{k_r+1,1}, s_{k_r+2,1}, \ldots, s_{k_{r+1},1},$$

respectively, that consist of *mutually weak bisimilar states*, i.e. any two states s_1 and s_2, such that $s_1 \in \{s_{j_r,1}, s_{j_r+1,1}, s_{j_r+2,1}, \ldots, s_{j_{r+1},1}\}$ and $s_2 \in \{s_{k_r,1}, s_{k_r+1,1}, s_{k_r+2,1}, \ldots, s_{k_{r+1},1}\}$, are weak bisimilar.

Definition 22 (Corresponding paths)**.** Let Q_1, Q_2 be weak bisimilar LTSs

1. Two infinite paths $\pi_1 \in Q_1$ and $\pi_2 \in Q_2$ such that

$$\pi_1 = (s_{0,1}, \alpha_{0,1}, s_{1,1})(s_{1,1}, \alpha_{1,1}, s_{2,1}) \ldots$$

$$\pi_1 = (s_{0,2}, \alpha_{0,2}, s_{1,2})(s_{1,2}, \alpha_{1,2}, s_{2,2}) \ldots$$

   π_1 and π_2 are called *corresponding paths*, i.e.

$$\pi_1 \approx \pi_2$$

   iff there exist two infinite sequences of indices $0 = j_0 < j_1 < j_2 < \ldots$ and $0 = k_0 < k_1 < k_2 < \ldots$ with $s_{j,1} \approx s_{k,2}$ for all $j_{r-1} \le j < j_r$, $k_{r-1} \le k < k_r$ with $r = 1, 2, \ldots$.

2. Two finite paths $\pi_1 \in Q_1$ and $\pi_2 \in Q_2$ such that

$$\pi_1 = (s_{0,1}, \alpha_{0,1}, s_{1,1}) \ldots (s_{j,1}, \alpha_{j,1}, s_{j+1,1})$$

$$\pi_1 = (s_{0,2}, \alpha_{0,2}, s_{1,2}) \ldots (s_{k,2}, \alpha_{k,2}, s_{k+1,2})$$

   π_1 and π_2 are called *corresponding paths*, i.e.

$$\pi_1 \approx \pi_2$$

   iff there exist two finite sequences of indices $0 = j_0 < j_1 < \ldots < j_l = K_1 + 1$ and $0 = k_0 < k_1 < \ldots < k_l = K_2 + 1$ with $s_{j,1} \approx s_{k,2}$ for all $j_{r-1} \le j < j_r$, $k_{r-1} \le k < k_r$ with $r = 1, 2, \ldots$.

Now we want to prove that if two states are weak bisimilar, i.e. $s_1 \approx s_2$, then for every path starting from s_1 there exists a corresponding path starting from s_2, and for every path starting from s_2 there exists a corresponding path starting from s_1.

Lemma 4.5.1. *Let $Q_1 = \langle S_1, \to_1, \iota_1, L_1\rangle$ and $Q_2 = \langle S_2, \to_2, \iota_2, L_2\rangle$ be weak bisimilar LTSs and $s_1 \in S_1$ and $s_2 \in S_2$. If $s_1 \approx s_2$ then $\forall \pi_1 \in Paths(s_1)$ $\exists \pi_2 \in Paths(s_2) : \pi_1 \approx \pi_2$.*

Here, $Path(s)$ denotes the set of maximal paths π with $first(\pi) = s$.

Before we start proving, we define a terminal state.

Definition 23 (Terminal state)**.** Let Q be an LTS and $Post(s, \alpha) = \{s' \in S | (s, \alpha, s')\}$. A state s is called *terminal* state if a set $Post(s)$ defined as $Post(s) = \bigcup_{\alpha \in AP} Post(s, \alpha)$ is empty.

Proof. Let $\pi_1 = (s_{0,1}, \alpha_{0,1}, s_{1,1})(s_{1,1}, \alpha_{1,1}, s_{2,1})\dots$ be a path from $Path(s_1)$. We define a corresponding path π_2 starting in s_2, where the transition $(s_{i,1}, \alpha_i, s_{i+1,1})$, here $\exists s_2' : s_{i,1} \approx s_2'$ and $s_{i+1,1} \not\approx s_2'$, are matched by transitions $(s_{i,2}, \tau, u_{i,1})\dots(u_{i,n_i}, \alpha_i, s_{i+1,2})$ such that $s_{i+1,1} \approx s_{i+1,2}$ and for all $k : 1 \le k \le n_i$ $s_{i,1} \approx u_{i,k}$.

The proof is by induction on i. For each case we distinguish between s_i being terminal state or not.

Basic of induction $i = 0$. If s_1 is a terminal state, it follows directly from $s_1 \approx s_2$ that either s_2 is a terminal too, or there exists a path $(s_2, \tau, u_{0,1})(u_{0,1}, \tau, u_{0,2})\dots$. Thus, the path π_2 consists of either one state s_2 or a path fragment $(s_2, \tau, u_{0,1})(u_{0,1}, \tau, u_{0,2})\dots$. If s_1 is not a terminal state, it follows from $s_1 \approx s_2$ that the transition $(s_1, \alpha_0, s_{1,1})$ can be matched by a transition

$$(s_2, \tau, u_{0,1})(u_{0,1}, \tau, u_{0,2})\dots(u_{0,n_0}, \alpha_0, s_{1,2}) \tag{4.4}$$

such that $s_{1,1} \approx s_{1,2}$. This yields the path fragment 4.4 be the path π_2.

Now $i > 0$ and we assume that the path fragment from s_2 is already constructed:

$$(s_{0,2}, \tau, u_{0,1})(u_{0,1}, \tau, u_{0,2})\dots(u_{0,n_0}, \alpha_0, s_{1,2})$$

$$(s_{1,2}, \tau, u_{1,1})(u_{1,1}, \tau, u_{1,2})\dots(u_{1,n_1}, \alpha_1, s_{2,2})\dots(u_{i-1,n_{i-1}}, \alpha_{i-1}, s_{i,2}) \tag{4.5}$$

and $s_{i,1} \approx s_{i,2}$. If $s_{i,1}$ is a terminal state, then there exists a path fragment

$$(s_{i,2}, \tau, u_{i,1})(u_{i,1}, \tau, u_{i,2})\dots \tag{4.6}$$

Thereof, the path π_2, which is a result of concatenating the path fragment 4.5 from $s_{0,2}$ to $s_{i,2}$ and the path fragment 4.6, fullfills the desired condition.

In the following, we assume that $s_{i,1}$ is not a terminal state, which means that π_1 does not end in the state $s_{i,1}$. There are two cases:

1. $(s_{i,1}, \alpha, s_{i+1,1})$ (visible step from $s_{i,1}$). Due to $s_{i,1} \approx s_{i,2}$ there exists a path fragment
$$(s_{i,2}, \tau, u_{i,1}) \dots (u_{i,n_i}, \alpha_i, s_{i+1,2}) \tag{4.7}$$
such that $s_{i+1,1} \approx s_{i+1,2}$ and $s_{i,1} \approx u_{i,1} \approx \dots \approx u_{i,n_i}$. Concatenating the path 4.5 with the path fragment 4.7 yields a path fragment that fulfilles the desired conditions.
2. $(s_{i,1}, \tau, s_{i+1,1})$ (invisible step from $s_{i,1}$). Then $s_{i+1,1} \approx s_{i,2}$ and we repeat our reasoning for the state s_{i+1} (the index for s_{i+1} is considered to be i again, but the state $s_{i,2}$ is still the same with the same index).

The resulting path fragment π_2 is a corresponding path to π_1.

□

In the following, we perform a key theorem, which states that the same ACTL formulas hold for two weak bisimilar LTSs.

Theorem 4.5.2. Let Q_1 and Q_2 be two weak bisimilar LTSs. Then Q_1 and Q_2 are ACTL equivalent.

Proof. Let $s_1 = \iota_{Q_1}$ and $s_2 = \iota_{Q_2}$ be initial states with $s_1 \approx s_2$ and let φ be an ACTL formula. Let $Q_1 \models \varphi$, we need to show that $Q_2 \models \varphi$.

$Q_1 \models \varphi$ means that $\forall \rho_1$ from Q_1 such that $first(\rho_1) = s_1$, $\rho_1 \models \varphi$. Due to the definition of weak bisimilar LTSs, we have $s_1 \approx s_2$. Due to Lemma 4.5.1 for every path π_1 from s_1 there exists a corresponding path π_2 from s_2, and vice versa. Therefore, if we show that a formula φ is satisfied by two runs $\rho_1 = (s_1, \pi_1)$ and $\rho_2 = (s_2, \pi_2)$, we show that $\forall \rho_2$ from Q_2 such that $first(\rho_2) = s_2$, $\rho_2 \models \varphi$.
Further, we prove by induction on the structure of φ that $\rho_1 \models \varphi \Leftrightarrow \rho_2 \models \varphi$

Basis of induction: $\varphi = A$, where $A \in$**AP**. By the definition of $\approx$, $\rho_1 \models A$ iff $\exists k : k \geq 1$ such that $\forall i : 1 \leq i < k$ (s_i, τ, s_{i+1}), i.e. first k transitions are labelled as "tau" in the path π_1, and $A \in L(\pi^k)$. Due to definition of weak bisimilar states, there exists a state $s'_2 \in S_2$ such that $s_2 \overset{\alpha}{\Longrightarrow} s'_2$. It means that $\rho_2 \models A$.
Further we consider several cases:

- $\varphi = \neg f$ such that $\rho_1 \models \neg f$ iff not $\rho_1 \models f$, this is by induction hypothesis equivalent to not $\rho_2 \models f$, which in turn is equivalent to $\rho_2 \models \neg f$.
- $\varphi = f \wedge g$ that means $\rho_1 \models f$ and $\rho_1 \models g$, this is by induction hypothesis is equivalent to $\rho_2 \models f$ and $\rho_2 \models g$, that implies $\rho_2 \models f \wedge g$.
- $\varphi = \exists f$. Suppose that $\rho_1 \models \exists f$, then there is a path, π_1 starting with s_1 such that $(s_1, \pi_1) \models f$. By Lemma 4.5.1, there is a corresponding path π_2 in Q_2 starting with s_2. By induction hypothesis, $(s_1, \pi_1) \models f$ iff $(s_2, \pi_2) \models f$. Therefore, $(s_1, \pi_1) \models \exists f$ implies $(s_2, \pi_2) \models \exists f$. The other direction is symmetric.

- $\varphi = f\, U\, g$. Suppose that $\rho_1 \models f\, U\, g$. By the definition of the until operator, there is a k such that $(s_{k,1}, \pi_1^k) \models g$ (here, the first bottom index means an ordinal number of a state in a path, the second bottom index, which is either 1 or 2, means that a state belongs to the LTS Q_1 or Q_2, the upper index means an index of a state, where it starts from) and for all $1 \leq j < k$, $(s_{j,1}, \pi_1^j) \models f$. Since π_1 and π_2 correspond, so do for π_1^j and π_2^j for any j. Therefore, by the inductive hypothesis, $(s_{m,2}, \pi_2^m) \models g$ and $(s_{j,2}, \pi_2^j) \models f$ for all $1 \leq j < m$. Therefore, $\rho_2 \models \varphi$. We can use the same argument for the other direction.
- $\varphi = Xf$. Suppose that $\rho_1 \models Xf$. By the definition of the nexttime operator, $\exists k \geq 2$ such that $(s_{k,1}, \pi_1) \models f$, $\forall i : 1 \leq i < k-1$ $(s_{i,1}, \tau, s_{i+1,1})$ and $\tau \notin L(\pi_1^{k-1})$. Due to $s_{1,1} \approx s_{1,2}$ and $s_{1,1} \overset{\alpha}{\Longrightarrow} s_{k,1}$, there exists $s_{m,2} \in S_2$ with $s_{1,2} \overset{\alpha}{\Longrightarrow} s_{m,2}$ such that $s_{k,1} \approx s_{m,2}$. Since π_1 and π_2 are corresponding paths, it means that the paths can be divided into segments of weak bisimilar states, where $s_{k,1} \approx s_{m,2}$. Therefore, by inductive hypothesis if $(s_{k,1}, \pi_1) \models f$, so $(s_{m,2}, \pi_2) \models f$, i.e. $\rho_2 \models Xf$.

□

4.5.3 Additional Theorem about ACTL Formulas Preservation

In this subsection we want to prove an additional fact that is latter used for interpretation of properties. We consider two sets $Set_1 \subseteq S_1$ and $Set_2 \subseteq S_2$, which consist of mutually weak bisimilar states (S_1 and S_2 are sets of states in weak bisimilar LTSs Q_1 and Q_2). Both sets Set_1 and Set_2 are maximal, i.e. there exists no state $s \notin S_1$ such that $s \approx s_2 \in S_2$ (the same statement holds for any state s from the set S_2). We want to prove that if we add a loop transition with label $\kappa \notin L$ for every state from S_1 and S_2, i.e. a transition of type (s_i, κ, s_i), then the extended LTSs stay weak bisimilar.

In the following, we define formally weak bisimilar sets of states.

Definition 24 (Weak bisimilar sets)**.** Let $Q_i = \langle S_i, \rightarrow_i, \iota_i, L\rangle$, for $i = 1, 2$, be two weak bisimilar LTSs, i.e. $Q_1 \approx Q_2$. Sets Set_1 and Set_2 are *weak bisimilar sets* if for every $s_1 \in Set_1$ and every $s_2 \in Set_2$ the following three statements hold:
(1) $s_1 \approx s_2$,
(2) there exists no state $s : s \in S_1$ and $s \notin Set_1$ such that $s_1 \approx s_2$,
(3) there exists no state $s : s \in S_2$ and $s \notin Set_2$ such that $s_1 \approx s_2$.

An optimal algorithm for building weak bisimilar sets is out of scope of this thesis. Here, we assume that when a state s in the LTSs Q_1 (Q_2) is chosen, all states in the LTSs Q_2 (Q_1) are analysed, if an analysed state is bisimilar with the chosen state s, it is appended to the set Set_2 (Set_1). Then the states of the LTS Q_1 (Q_2) are analysed. If a state from Q_1 (Q_2) is weak bisimilar with any state from the set Set_2 (Set_1), then it is appended to the set Set_1.

Note that if we have two weak bisimilar LTSs $Q_i = \langle S_i, \rightarrow_i, \iota_i, L\rangle$, for $i = 1, 2$, then for every state in one of these LTSs there exist nonempty weak bisimilar sets $Set_1 \subseteq S_1$ and $Set_2 \subseteq S_2$. The fact directly follows from the definition of weak bisimilar LTSs.

In the following theorem we want to show that if we add an additional loop transition to every state from weak bisimilar sets Set_1 and Set_2, then the extended LTSs stay weak bisimilar.

Theorem 4.5.3. Let $Q_i = \langle S_i, \rightarrow_i, \iota_i, L\rangle$, for $i = 1, 2$, be two weak bisimilar LTSs, i.e. $Q_1 \approx Q_2$. Let Set_1 and Set_2 be weak bisimilar sets of states ($Set_1 \subseteq S_1$ and $Set_2 \subseteq S_2$). If we extend the set of labels with a label κ, i.e. $L' = L \cup \{\kappa\}$ ($\kappa \notin L$), and the set of loop transitions such that

$$\rightarrow'_1 = \rightarrow_1 \cup \sum (s_{i,1}, \kappa, s_{i,1})$$

and

$$\rightarrow'_2 = \rightarrow_2 \cup \sum (s_{j,2}, \kappa, s_{j,2})$$

for all $s_{i,1} \in Set_1$ and all $s_{j,2} \in Set_2$. Then the extended LTSs $Q'_1 = \langle S_1, \rightarrow'_1, \iota_1, L'\rangle$ and $Q'_2 = \langle S_2, \rightarrow'_2, \iota_2, L'\rangle$ are weak bisimilar.

Proof. We need to show that the conditions of Definition 17 hold for Q'_1 and Q'_2. Due to the fact that $Q_1 \approx Q_2$, the conditions hold for every state $s_1 \in S_1$ and $s_2 \in S_2$ with a transition with a label $\alpha \in L$. Then we need to consider transitions with a label $\kappa \in L'$ ($\kappa \notin L$). Let s_1 be a state from S_1 and there exists a transition (s_1, k_1, s_1) (the proof for the case $s_2 \in S_2$ is similar). Due to the fact that there exist two weak bisimilar sets of states Set_1 and Set_2 in Q_1 and Q_2, respectively, such that $s_1 \in Set_1$, and the fact that $\forall s_j \in Set_2\ \exists (s_j, k, s_j)$, the conditions of Definition 17 hold for a random state $s_1 \in S_2$. $\square$

4.6 Summary

In this chapter, we defined the weak bisimulation over LTSs. Then, we introduced an action-based logic, called ACTL, which is an equivalent to a state-based logic CTL*. We explained how to specify and verify behavioural properties with ACTL for a graph language. Finally, we proved that weak bisimulation implies ACTL equivalence, i.e.

$$\text{if } Q_1 \approx Q_2 \text{ then } \forall \varphi \in ACTL \text{ follows that } Q_1 \models \varphi \text{ iff } Q_2 \models \varphi$$

It means that if we define a model transformation over two modelling languages, for models of which it is possible to generate an LTS, then we can check a behavioural correctness (of the model transformation) by comparison of respective LTSs. If it is possible to establish a weak bisimulation over an LTS generated for every possible source model and an LTS generated for a target model, which is a result of a model transformation, it means that the model transformation preserves all behavioural properties (specified with ACTL) of a source model. The weak bisimulation must be then defined over states of LTSs. Recall that in our case we deal with languages defined by graph transformations, it means that a state in an LTS is a graph. In the next chapter we introduce a method, which provides instructions how to define an equivalence relation over all possible graphs typed over run-time graphs and to prove that the defined equivalence relation is a weak bisimulation. Such method guarantees the preservation of behavioural properties specified with ACTL.

CHAPTER 5

Method for Semantics Preserving Model Transformation

We consider the property of semantics preservation to be particularly important for model transformation. The semantics preservation in the context of this thesis means that a generated target model has the same behaviour as a source model. The same behaviour means that a target model has the same behavioural properties as a source model. In this chapter, we introduce a method[1] to ensure that *every* target model has the same behaviour as the original source model (see Figure 5.1) and interpretation of behavioural properties for the transformed model.

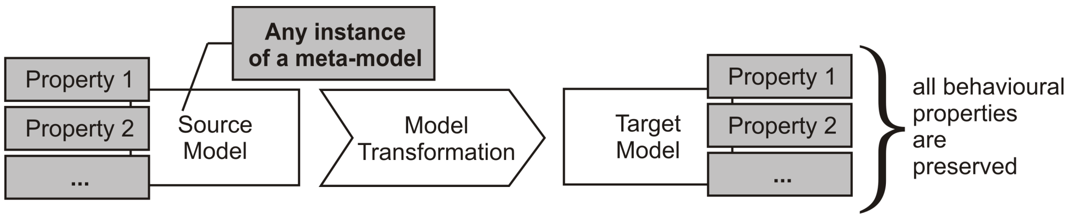


Figure 5.1: Semantics preserving model transformation

[1]The method is explained on the example of the model transformation and the proof of its correctness with two simple (self-defined) languages. The example of model transformation and the proof are originally taken from [HKR$^+$10b]

The problem of semantics preservation of model transformation is not solved in the MDA approach, where model transformation plays an important role for the software development process (see Chapter 2). The MDA relies on the stepwise development process where the platform independent model is transformed either into a platform specific model or into another model at the same abstraction layer. The correctness of model transformation is crucial for MDA, because it is important not to change or misinterpret the initial requirements during the software development process.

In this chapter, we describe a method that ensures the *full* behavioural semantics preservation of model transformation in the context of MDA. "Full" means that the method ensures the behaviour preservation not only for a single model, but for *any* model. The advantage of using our method is that the behavioural preservation is proven once for model transformation and then the behavioural preservation holds for *every* target model automatically, when this model transformation is applied.

The method deals with languages, the behavioural semantics of which can be formally specified by means of graph transformation systems (see Chapter 3). The graph transformation systems give rise to a transition system modelling its execution. This in turn allows to apply standard concepts from concurrency theory [HM85] which let us decide whether the transition systems are indeed equivalent or not. Our aim is eventually to show an equivalence called *weak bisimilation* between the transition systems of a source model and a target model. The weak bisimulation implies the preservation of behavioural properties (see Chapter 4).

5.1 Problem Definition

We try to solve the problem of behavioral semantics preservation for model transformation. We use graph transformations as a formalism to specify the modelling languages and a model transformation. Therefore, we consider modelling languages as *graph languages*, i.e. sets of graphs, and the models are the graphs themselves. The model transformation is a rule system which consists of graph transformation rules. Formally, there are two graph languages L_1 and L_2. A Labelled Transition System (LTS) is a model that describes the behavioural semantics of a graph language, which will henceforth be denoted as $Q(G)$, where G is a graph of a graph language L. We want to specify a model transformation of a graph G_1 of L_1 into a graph G_2 of L_2. We want that for a behavioural property, formalized as φ, and its interpretation χ in the language L_2 the following statement holds:

$$Q(G_1) \models \varphi \quad \Rightarrow Q(G_2) \models \chi(\varphi)$$

In addition, we require:

- the model transformation to be applicable to every model of the source language,

- the proof of behaviour preservation during model transformation must hold for *every* model of the source language,
- it must be possible to specify the property φ by means of ACTL (see Chapter 4, Section 4.4).
- the meaning of the function χ must be explained, i.e. the way how behavioural property φ, which is specified for G_1 in L_1, is interpreted for G_2 in L_2.

We proceed as follows. Firstly, we give an introduction to our method, where we define the requirements for modelling languages and explain how the criteria for behavioural preservation could be shown. Then, we use sample languages to illustrate how to specify model transformation in terms of graph transformations. Thereafter, we show how to establish the weak bisimilar relation between the transition system of *any* source model and that of the target model resulting from its transformations. Later, we discuss the method results. The final section contains an analysis of the method.

5.2 Proposed Solution

We propose a method to ensure the behavioural preservation during a model transformation. Our method includes the language restrictions that we intend to work with, a specification of model transformation and the establishment of an equivalence criteria that guarantees the behavioural preservation during model transformations. Further, we describe individually each step of the solution, which is also depicted on Figure 5.2 as an enriched number.

Step 1 The first step of our method is the definition of the abstract syntax (later syntax) for the modelling languages. According to the MDA approach the syntax of a modelling language is defined with a meta-model. Since we decided on a graph transformation (see Chapter 3), we require the syntax of a modelling language be defined with a type graph (see Chapter 3, Section 3.1).

Step 2 We work only with languages for which the behavioural semantics is defined. On the second step we require the behavioural semantics be formally specified by means of graph transformation rules over a run-time type graph (see Chapter 3, Section 3.3).

Step 3 The non-trivial mapping between the graph transformation rules is needed in order to show the correctness of model transformation with respect to behavioural preservation. The mapping is based on the knowledge of the behaviour semantics, i.e., some graph transformation rules describe the same behaviour of elements that are typed over the run-time type graphs. By this assuming, the corresponding graph transformation rules (the rules that describe similar behaviour) are mapped. Thus, some rules perform the steps in the source transition system that have a match in the target transition system and vice versa. We say that the mapped rules perform *observable steps* in the

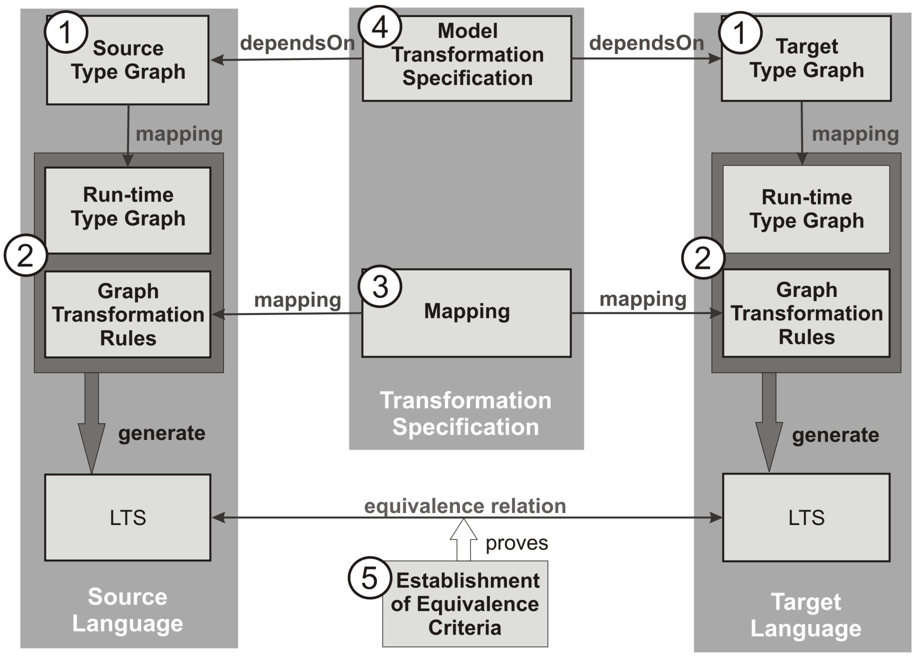


Figure 5.2: Overview of the proposed solution, which includes restrictions on the languages (1)-(3), a specification of model transformation (4) and a criteria for the behavioural correctness of model transformation (5)

LTS. Unmapped rules fulfill *internal steps*, in the sense that the rules could not be seen from the other label transition system, since they have no match. Due to the transition set of LTSs is generated by the graph transformation rules and labelled over the rule names set, the mapping allows to compare two different LTSs.

Step 4 In the fourth step of our method we require the existence of a non-trivial mapping between syntactic elements of the source and the target languages. This mapping is essential for establishing a model transformation, because the model transformation themselves specify the transformation of syntax. We also require the model transformation to be done in a TGG style (see Chapter 3, Section 3.5). The TGG rules are defined over a type graph, which consists of the type graphs (defined in step 1) and a graph which points on the corresponding syntactic elements. The TGG rules must keep these correspondences.

Step 5 In the fifth step we want to show the correctness of the model transformation. For this, we compare the LTSs generated by application of graph transformation rules (defined in step 2) to the models received as a result of the model transformation (defined in step 4). We compare the LTSs with respect to the mapping (defined in step 3). We establish the equivalence relation

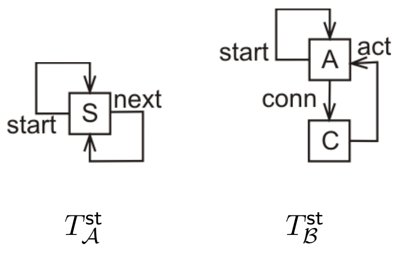

start
S
next
start
A
act
conn
C
$T^{\mathsf{st}}_{\mathcal{A}}$
$T^{\mathsf{st}}_{\mathcal{B}}$

Figure 5.3: Syntactic (st) type graphs for graph languages $\mathcal{A}$ and $\mathcal{B}$

over the LTSs and prove that this relation is a weak bisimulation.

If there are languages with the specified restrictions in steps (1)-(3) then we can define a model transformation and ensure their correctness.

5.3 Method

The model transformation and the proof of its correctness is shown in an example with two simple (self-defined) languages. Firstly, we specify the syntax of the languages with a type graph (from Chapter 3, Section 3.2 we know that a type graph functions as a meta-model and typed graphs are models) and a behavioural semantics of languages with graph transformations (Chapter 3, Section 3.4). After this, we define a mapping function that specifies the corresponding semantic rules. Later, we construct the model transformations between the two languages. Finally, we show how to establish equivalence criteria over the LTSs.

The steps performed in this section are the steps (1)-(5) of the method described above to show behaviour preservation of model transformation.

5.3.1 Language Syntax (Step 1)

Our running example consists of two distinct, very simple graph languages denoted $\mathcal{A}$ and $\mathcal{B}$. Figure 5.3 shows type graphs for the languages, denoted $T^{\mathsf{st}}_{\mathcal{A}}$ and $T^{\mathsf{st}}_{\mathcal{B}}$, respectively. They describe the typing of the *syntactic* parts of our two languages.

The type graphs themselves impose only a weak structure: not all graphs that can be typed over the $\mathcal{A}$- and $\mathcal{B}$-type graphs are considered to be part of the languages. Instead, we impose the following further constraints on the syntactic structure:

Language $\mathcal{A}$ consists of next-connected S-labelled nodes (*statements*). There should be a single S-node with a start-edge to itself (we say also a start-loop), from which all other nodes are reachable (via paths of next-edges). Furthermore, no next-loops are allowed.

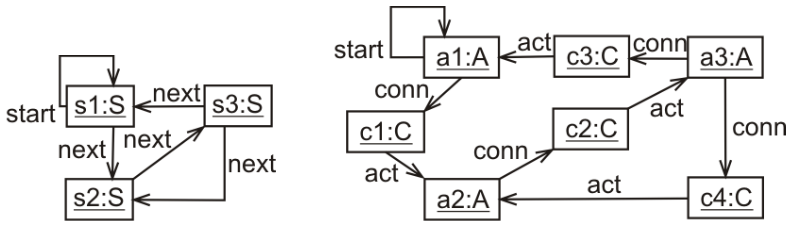


Figure 5.4: Example graphs of languages $\mathcal{A}$ (in the left) and $\mathcal{B}$ (in the right)

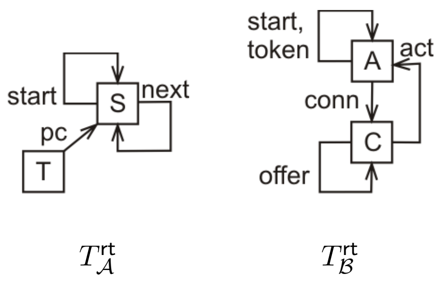

start
S
next
pc
T
start,
token
A
act
conn
C
offer
$T^{\mathsf{rt}}_{\mathcal{A}}$
$T^{\mathsf{rt}}_{\mathcal{B}}$

Figure 5.5: Run-time(rt) type graphs for graph languages $\mathcal{A}$ and $\mathcal{B}$

Language $\mathcal{B}$ consists of bipartite graphs of A- (*action*) and C-labelled (*connector*) nodes. Every C-node has exactly one incoming conn-edge and exactly one outgoing act-edge; the opposite nodes of those edges must be distinct. Like $\mathcal{A}$-graphs, $\mathcal{B}$-graphs have exactly one node with a start-self-edge, from which all other nodes are reachable (via paths of conn- and act-edges).

Small example graphs of instances of languages $\mathcal{A}$ and $\mathcal{B}$ are shown in Figure 5.4. We use $\mathcal{G}^{\mathsf{st}}_{\mathcal{A}}$ ($\mathcal{G}^{\mathsf{st}}_{\mathcal{B}}$) to denote the set of all well-formed (syntactic) $\mathcal{A}$-graphs ($\mathcal{B}$-graphs).

5.3.2 Language Semantics (Step 2)

We specify the behavioural semantics by means of *graph transformation rules*. This means that the graphs will represent run-time states. As we will see, this will involve auxiliary node and edge types that do not occur in the language type graphs. Figure 5.5 shows extended type graphs $T^{\mathsf{rt}}_{\mathcal{A}}$ and $T^{\mathsf{rt}}_{\mathcal{B}}$ that include these *run-time* types. For $\mathcal{A}$, a T-node (of which there can be at most one) models a *thread*, through a single program counter (pc-labelled edge). For $\mathcal{B}$, we use token- and offer-loops which play a similar role; details will become clear below. Similar to the syntactic part, we use $\mathcal{G}^{\mathsf{rt}}_{\mathcal{A}}$ ($\mathcal{G}^{\mathsf{rt}}_{\mathcal{B}}$) to denote the set of well-formed (run-time) $\mathcal{A}$-graphs ($\mathcal{B}$-graphs). The semantics of $\mathcal{A}$- and $\mathcal{B}$-models is defined in Figure 5.6.

The graph transformation rule systems is as defined in Definition 11. We let $dom(\mathcal{RS}_{\mathcal{A}}) = \{\mathsf{initA}, \mathsf{movePC}\}$ and $dom(\mathcal{RS}_{\mathcal{B}}) = \{\mathsf{initB}, \mathsf{createO}, \mathsf{moveT}\}$ be the

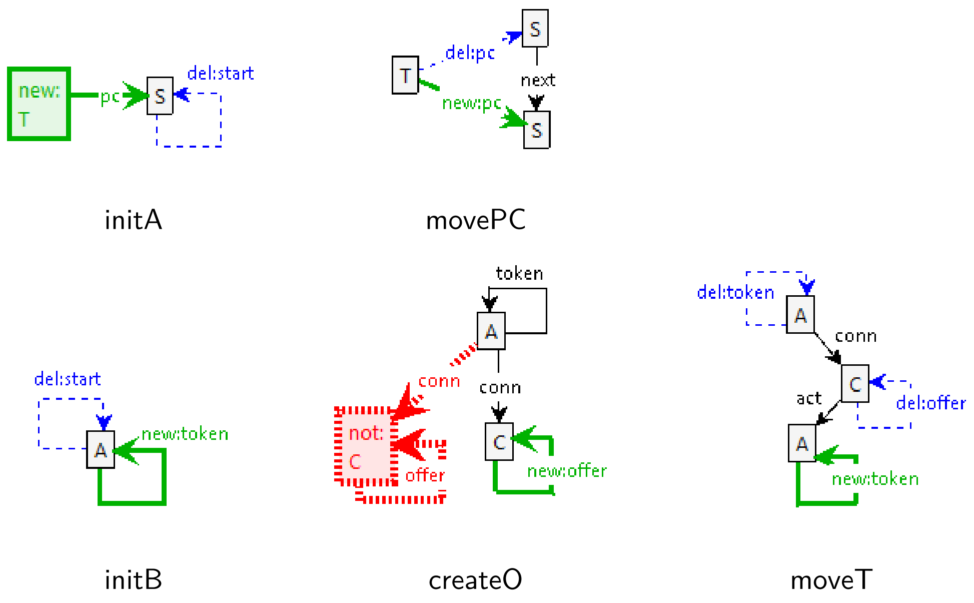


Figure 5.6: Behavioural semantics rules for $\mathcal{A}$ (initA and movePC) and $\mathcal{B}$ (initB, createO and moveT)

names in the rule systems for the $\mathcal{A}$- and $\mathcal{B}$-models (see Figure 5.6). Intuitively, the init-rules perform an initialisation of the run-time system, setting the program counter to the start statement (in $\mathcal{A}$) or putting a token onto a start action (in $\mathcal{B}$). Rule movePC simply moves the program counter to the next statement, createO moves an offer to a C-node and moveT moves the token. The semantics of $\mathcal{A}$- and $\mathcal{B}$-graphs is completely fixed by these rules, giving rise to an LTS (Chapter 3 Section 3.4) summarizing all these executions.

5.3.3 Mapping over the Rule Systems (Step 3)

Our objective is to compare the LTSs of graphs of languages $\mathcal{A}$ and $\mathcal{B}$. In Section 5.3.4 we will define model transformation $MT \subseteq \mathcal{G}^{\mathsf{st}}_{\mathcal{A}} \times \mathcal{G}^{\mathsf{st}}_{\mathcal{B}}$ translating $\mathcal{A}$-graphs to $\mathcal{B}$-graphs. We aim at proving this model transformation to be *behavioural semantics preserving*, in the sense that the LTSs of source and target models are always weak bisimilar.

However, there is an obvious problem: the LTSs of $\mathcal{A}$- and $\mathcal{B}$-graphs do not have the same labels, in fact $dom(\mathcal{RS}_{\mathcal{A}}) \cap dom(\mathcal{RS}_{\mathcal{B}}) = \emptyset$. Nevertheless, there is a clear intuition which rules correspond to each other: on the one hand the two initialisation rules, and on the other hand the rules movePC and createO. The reason for taking the latter two as corresponding is that both rules decide on where control is moving. The rule moveT has no matching counterpart in the $\mathcal{A}$-language, it can be seen as an *internal* step of the $\mathcal{B}$-language, completing a step initiated by createO. These observations give rise to the following mappings defined on the labels (i.e., the rule names) to a common set of names.

$$map_{\mathcal{A}}: \quad \mathsf{initA} \mapsto \mathsf{init}, \quad \mathsf{movePC} \mapsto \mathsf{move}$$

$$map_{\mathcal{B}}: \quad \mathsf{initB} \mapsto \mathsf{init}, \quad \mathsf{createO} \mapsto \mathsf{move}, \quad \mathsf{moveT} \mapsto \tau$$

Let be a set $\mathsf{Sym} = \{\mathsf{init}, \mathsf{move}, \tau\}$ a common set of names for the rules from $dom(\mathcal{RS}_{\mathcal{A}})$ and $dom(\mathcal{RS}_{\mathcal{B}})$, here τ stays for the internal steps. We call such a mapping
$map\colon dom(\mathcal{RS}) \to \mathsf{Sym}$ (for a given rule system $\mathcal{RS}$) *non-trivial* if it does not map every rule name to τ.

In order to provide a definition of semantics preserving model transformations, we define a mapping *map* also over the LTSs: $map\colon \mathcal{P}(Q) \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is a universe of LTSs. The mapping functions as following, let Q be an LTS, such that $Q = \langle S, \to, i, L\rangle$, then $map(Q) = \langle S, \to', i, L'\rangle$, where $\to' = \{(s, map(l), s') \mid (s, l, s') \in \to\}$ and $L' = \{map(l) \mid l \in L\}$.

Our ultimate goal is to show that our model transformation MT is semantics preserving, it means that for a behavioural property, formalized as φ, and its interpretation χ in the $\mathcal{B}$-language the following statement holds:

$$Q(G_{\mathcal{A}}) \models \varphi \quad \Rightarrow \quad Q(G_{\mathcal{B}}) \models \chi(\varphi) \tag{5.1}$$

However, we present a method that allows to assure a more stronger statement:

$$map_{\mathcal{A}}(Q(G_{\mathcal{A}})) \approx map_{\mathcal{B}}(Q(G_{\mathcal{B}})) \tag{5.2}$$

here $G_{\mathcal{A}} \in \mathcal{G}^{st}_{\mathcal{A}}$, $G_{\mathcal{B}} \in \mathcal{G}^{st}_{\mathcal{B}}$ with $MT(G_{\mathcal{A}}, G_{\mathcal{B}}) \subseteq \mathcal{G}^{st}_{\mathcal{A}} \times \mathcal{G}^{st}_{\mathcal{B}}$. The mapping functions $map_{\mathcal{A}}\colon dom(\mathcal{RS}_{\mathcal{A}}) \to \mathsf{Sym}$ and $map_{\mathcal{B}}\colon dom(\mathcal{RS}_{\mathcal{B}}) \to \mathsf{Sym}$ are non-trivial functions. $\approx$ denotes weak bisimulation.

The important fact is that Condition 6.1 implies Condition 5.1 (see Chapter 4 for more information). Later we call our model transformation *MT semantics preserving*, if Condition 6.1 holds.

5.3.4 Model Transformation (Step 4)

Our model transformation needs to translate $\mathcal{A}$-models into $\mathcal{B}$-models. We use triple graph grammars (TGGs) (see Chapter 3, Section 3.5) which are well-suited for defining model transformation. The idea of TGG is that the graphs can be separated into three subgraphs, each being typed over its own type graph. Two

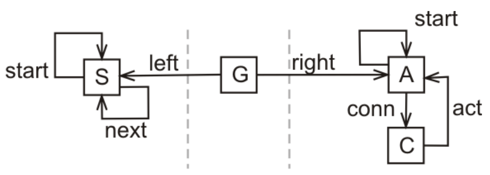


Figure 5.7: Type graph $T^{\mathsf{st}}_{\mathcal{AB}}$ for TGG graph rules

of these subgraphs evolve simultaneously while the third keeps correspondences between them.

For our example, we have the two type graphs $T^{\mathsf{st}}_{\mathcal{A}}$ and $T^{\mathsf{st}}_{\mathcal{B}}$ which – for forming a type graph for TGGs – are conjoined and augmented with one new correspondence G-node (the glue) (see Figure 5.7). This combined type graph is denoted $T^{\mathsf{st}}_{\mathcal{AB}}$.

The TGG style is different to any other style of model transformation. In contrast to a transformation, when the source model is given in the beginning and is then gradually transformed, TGG rules build two models simultaneously, relating each part of the source model to the target one.

There is no deletion in our TGG transformations and no particular constraints. Therefore, when the needed source model is built, the model transformation construction terminates. The output is a model which consists of three models: a source model, a corresponding model and a target model. This allows to keep correspondences between transformed elements and to prove certain properties of the corresponding graphs. The TGG rules for the $\mathcal{A}$ to $\mathcal{B}$ transformation are given in Figure 5.8.

These rules incrementally build combined $\mathcal{A}$ and $\mathcal{B}$-graphs. Initially, only the upper rule in Figure 5.8 can be applied and its application constructs a graph with one S- and one A-node connected via one correspondence node. The middle rule allows to create further S, A and C-nodes together with their correspondences, and the lower rule simultaneously generates new next-edges between S-nodes and connections between A-nodes via C nodes, however only for *corresponding* S- and A-nodes. We let $\mathcal{G}^{\mathsf{st}}_{\mathcal{AB}}$ denote the set of graphs obtained by applying the three TGG rules on an empty start graph. To obtain the translation at the end, we need to project the final graph onto the type graphs of $\mathcal{A}$ and $\mathcal{B}$. Using the definition of projection as given in Chapter 3, Section 3.1, the model transformation $MT_{\mathcal{A}2\mathcal{B}}$ thus works as follows: Given an $\mathcal{A}$-graph $G_{\mathcal{A}}$ and a $\mathcal{B}$-graph $G_{\mathcal{B}}$, we have $MT_{\mathcal{A}2\mathcal{B}}(G_{\mathcal{A}}, G_{\mathcal{B}})$ exactly if there is some $G_{\mathcal{AB}} \in \mathcal{G}^{\mathsf{st}}_{\mathcal{AB}}$ such that $G_{\mathcal{A}} = \pi_{T^{\mathsf{st}}_{\mathcal{A}}}(G_{\mathcal{AB}})$ and $G_{\mathcal{B}} = \pi_{T^{\mathsf{st}}_{\mathcal{B}}}(G_{\mathcal{AB}})$.

5.3.5 Establishment of Weak Bisimulation (Step 5)

We want to show that the previously defined model transformation $MT_{\mathcal{A}2\mathcal{B}}$ are semantics preserving. For this, we compare the LTSs that are generated by the rule systems $\mathcal{RS}_{\mathcal{A}}$ and $\mathcal{RS}_{\mathcal{B}}$ by establishing a relation $\mathcal{R}$ over them. We consider

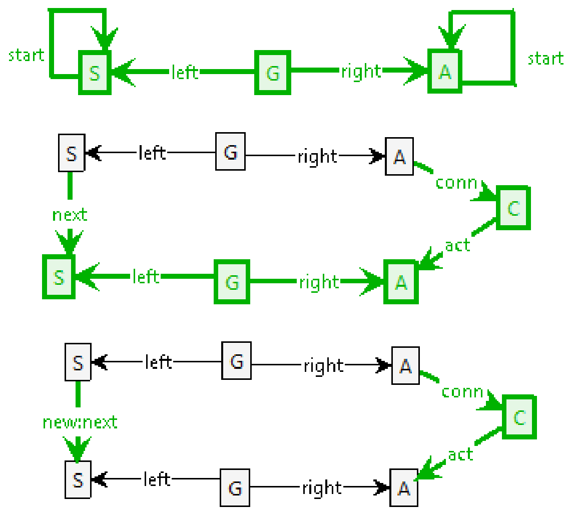


Figure 5.8: TGG transformation rules, which define model transformation $MT_{\mathcal{A}2\mathcal{B}}$

the mappings $map_{\mathcal{A}}$ and $map_{\mathcal{B}}$. By proving that the defined relation $\mathcal{R}$ is a *weak bisimulation*, we prove the correctness of the model transformation $MT_{\mathcal{A}2\mathcal{B}}$ with respect to behaviour preservation.

We start by short summarizing the results achieved in steps (1)-(4) and make some observations concerning the correspondences generated by TGG rules and the graph transformation rules that describe the behavioural semantics of the $\mathcal{A}$- and $\mathcal{B}$-languages. This preparatory reasoning helps us to define the relation $\mathcal{R}$ and to prove that the defined relation is a weak bisimulation.

In Section 5.3.4 we defined model transformation $MT_{\mathcal{A}2\mathcal{B}}$ that build a graph $G_{\mathcal{AB}} \in \mathcal{G}^{st}_{\mathcal{AB}}$, which consists of the graph $G_{\mathcal{A}}$, the graph $G_{\mathcal{B}}$ and the correspondences between them. We apply the graph transformation rules (defined in Figure 5.6) from the rule systems $\mathcal{RS}_{\mathcal{A}}$ and $\mathcal{RS}_{\mathcal{B}}$ to the separate graphs $G_{\mathcal{A}} = \pi_{T^{st}_{\mathcal{A}}}(G_{\mathcal{AB}})$ and $G_{\mathcal{B}} = \pi_{T^{st}_{\mathcal{B}}}(G_{\mathcal{AB}})$. There are two essential observations that can be made. The first is that the graph transformation rules from the rule systems $\mathcal{RS}_{\mathcal{A}}$ and $\mathcal{RS}_{\mathcal{B}}$ do not change the syntactic structure of graphs. The second is that although the graph transformation rules are applied to the separate models, we still take in account the correspondences generated by model transformation $MT_{\mathcal{A}2\mathcal{B}}$. We proceed with formalizing these observations.

Notation To write down formally our reasoning, we start by defining some notation. To formulate structural correspondences, we introduce the following notation. For an S-node v_S and an A-node v_A, we write $corr(v_S, v_A)$ if there is a G-node v_G and a left-edge from v_S to v_G and a right-edge from v_G to v_A. For an edge e labelled label going from a node v to v', we simply write label(v, v').

We also use these as predicates.

First observation Both for $\mathcal{A}$- and $\mathcal{B}$-models, the graph transformation rules from $\mathcal{RS}_\mathcal{A}$ and $\mathcal{RS}_\mathcal{B}$ keep the syntactic structure of a model, except for start-edges: all S-nodes and next-edges, and all A, C-nodes and conn, act-edges stay the same.

We formalize our first observation that the syntactic structure of graphs stays the same (except for start edges) when the graph transformation rules are applied.

Proposition 5.3.1. *Let $G_\mathcal{A} \in \mathcal{G}^{\mathsf{rt}}_\mathcal{A}$ be an $\mathcal{A}$-graph. If $G_\mathcal{A} \xrightarrow{r} G'_\mathcal{A}$ for some $r \in \mathcal{RS}_\mathcal{A}$ then $\pi_{\mathsf{T}^{\mathsf{s}}_\mathcal{A}\backslash\mathsf{start}}(G_\mathcal{A}) = \pi_{\mathsf{T}^{\mathsf{s}}_\mathcal{A}\backslash\mathsf{start}}(G'_\mathcal{A})$, where* T\start *is the type* T *without the* start-*edge.*
A corresponding property holds for $\mathcal{B}$.

A number of further observations show that (1) corresponding nodes either both or none have start-edges, and (2) next-edges between S-nodes will generate connections via C-nodes between corresponding A-nodes and vice versa.

Second observation Correspondences between nodes in $\mathcal{A}$-models and $\mathcal{B}$-models are kept during application of behavioural semantic rules. Predicate *corr* as well as 5.3.1 and properties (1) and (2) can thus also be applied to separate $\mathcal{A}$ and $\mathcal{B}$-graphs.

We continue further with the formalization of the second observation. We start with the result, which shows that correspondences between S and A-nodes are unique. Here, $\exists!$ stands for "there exists exactly one".

Proposition 5.3.2. *Let $G \in \mathcal{G}_{\mathcal{AB}}$, v_S an* S*-node and v_A an* A*-node in G. Then the following two properties hold:*
(A) $\exists! v$ of type A *such that $corr(v_S, v)$, and*
(B) $\exists! v$ of type S *such that $corr(v, v_A)$.*

The following propositions illustrate some additional correspondences (properties (1)-(2)) which can be shown by induction on the application of the TGG rules. The first concerns start-edges:

Proposition 5.3.3. *Let $G \in \mathcal{G}_{\mathcal{AB}}$, v_S an* S*-node, v_A an* A*-node and let $corr(v_S, v_A)$. Then*

$$v_S \textit{ has a } \mathsf{start}\textit{-edge} \quad \textit{iff} \quad v_A \textit{ has a } \mathsf{start}\textit{-edge.}$$

Moreover, there is exactly one start-edge on the $\mathcal{A}$- and one on the $\mathcal{B}$-side. The next correspondence properties hold between next-edges and connections via C-nodes.

Proposition 5.3.4. *Let $G \in \mathcal{G}_{\mathcal{AB}}$, v_S an* S*-node, v_A an* A*-node and let $corr(v_S, v_A)$.*

- *If there is a* C*-node* v_C*, such that* $\mathsf{conn}(v_A, v_C)$*, then there is an* S*-node* v'_S *and an* A*-node* v'_A *such that* $\mathsf{next}(v_S, v'_S)$*,* $\mathsf{act}(v_C, v'_A)$ *and* $corr(v'_S, v'_A)$*.*
- *If there is an* S*-node* v'_S *such that* $\mathsf{next}(v_S, v'_S)$*, then there is a* C*-node* v_C *and an* A*-node* v_A *such that* $\mathsf{conn}(v_A, v_C)$*,* $\mathsf{act}(v_C, v'_A)$ *and* $corr(v'_S, v'_A)$*.*

The formulated propositions are crucial parts of our proof that the model transformations $MT_{\mathcal{A}2\mathcal{B}}$ are semantics preserving. The propositions also help us to define the conditions on the syntactic structure of graphs in a relation $\mathcal{R}$.

On the next step we define the relation $\mathcal{R}$ (defining $\approx$) over the states of two LTSs, i.e., over the states of $Q(G^0_{\mathcal{A}})$ and $Q(G^0_{\mathcal{B}})$, which are generated by application of the graph transformation rules (defined in Section 5.3.2) from the rule systems $\mathcal{RS}_{\mathcal{A}}$ and $\mathcal{RS}_{\mathcal{B}}$ to the graphs $G_{\mathcal{A}} = \pi_{T^{\mathsf{st}}_{\mathcal{A}}}(G_{\mathcal{AB}})$ and $G_{\mathcal{B}} = \pi_{T^{\mathsf{st}}_{\mathcal{B}}}(G_{\mathcal{AB}})$, respectively. Since the states of our LTSs are graphs, the relation $\mathcal{R}$ is defined over the well-formed run-time graphs $\mathcal{R} \subseteq \mathcal{G}^{\mathsf{rt}}_{\mathcal{A}} \times \mathcal{G}^{\mathsf{rt}}_{\mathcal{B}}$. We want that the relation $\mathcal{R}$ consists of the pairs of graphs $G_{\mathcal{A}}$ and $G_{\mathcal{B}}$ that denote the weak bisimilar states of the compared LTSs (see the definition of weak bisimulation in Chapter 4). Thus, if $(G_{\mathcal{A}}, G_{\mathcal{B}}) \in \mathcal{R}$, then the behaviour of the graph $G_{\mathcal{A}}$ after we apply the rules from $\mathcal{RS}_{\mathcal{A}}$ is similar to the behaviour of the graph $G_{\mathcal{B}}$ after the application of the corresponding rules from $\mathcal{RS}_{\mathcal{B}}$, we allow also an application of the rules which perform internal steps.

In order to construct $\mathcal{R}$ we are guided by Propositions 5.3.1 and 5.3.3 and additional considerations about run-time properties on the corresponding nodes. The example of a run-time property is, if A-node and S-node are corresponding nodes then they are activated within their behavioural semantics systems simultaneously, e.g. there must be a pc-edge for the S-node and a token-edge for the A-node. So, we define all pairs of $\mathcal{A}$- and $\mathcal{B}$-graphs:

(1) The $\mathcal{A}$ and $\mathcal{B}$-graphs follow the syntactic structure (except for start) generated by the TGG rules (see Proposition 5.3.1).

(2) The $\mathcal{A}$ and $\mathcal{B}$-graphs have start-edges only on corresponding nodes (see Proposition 5.3.3).

(3) The $\mathcal{A}$ and $\mathcal{B}$-graphs must exhibit run-time properties only on corresponding nodes. Figure 5.9 further illustrates condition (3). We have two possibilites for run-time elements in matching states: either the pc-edge is on an S-node and the token is on the corresponding A-node and no further offers exist (on the left), or the pc-edge is on a node for which the corresponding A-node has no token yet, but an offer has already been created and is ready to move the token to the A-node by means of the invisible step moveT (on the right).

(4) The condition (4) must obey well-formedness criteria for run-time elements.

We formalize our observations.

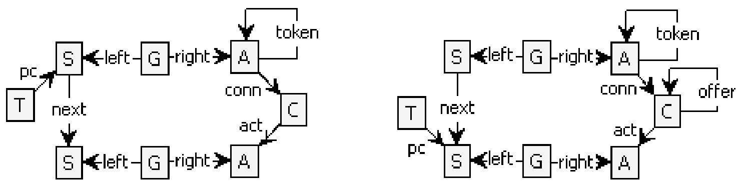


Figure 5.9: Illustration of condition (3): Left (i), right (ii)

$$
\begin{aligned}
\mathcal{R} \;=\; & \{(G_{\mathcal{A}}, G_{\mathcal{B}}) \in \mathcal{G}^{\mathsf{rt}}_{\mathcal{A}} \times \mathcal{G}^{\mathsf{rt}}_{\mathcal{B}} \mid \exists G_{\mathcal{AB}} \\
& (1)\ (\pi_{\mathsf{T}^{\mathsf{s}}_{\mathcal{A}} \setminus \mathsf{start}}(G_{A}) = \pi_{\mathsf{T}^{\mathsf{s}}_{\mathcal{A}} \setminus \mathsf{start}}(G_{\mathcal{AB}})) \wedge (\pi_{\mathsf{T}^{\mathsf{s}}_{\mathcal{B}} \setminus \mathsf{start}}(G_{\mathcal{B}}) = \pi_{\mathsf{T}^{\mathsf{s}}_{\mathcal{B}} \setminus \mathsf{start}}(G_{\mathcal{AB}})), \\
& (2)\ \forall\ \mathsf{S}\text{-nodes } v_S \text{ in } G_{\mathcal{A}},\ \mathsf{A}\text{-nodes } v_A \text{ in } G_{\mathcal{B}} \text{ s.t. } corr(v_S, v_A): \\
& \qquad \mathsf{start}(v_S) \text{ iff } \mathsf{start}(v_A), \\
& (3)\ \forall\ \mathsf{S}\text{-nodes } v_S \text{ in } G_{\mathcal{A}},\ \mathsf{A}\text{-nodes } v_A \text{ in } G_{\mathcal{B}} \text{ s.t. } corr(v_S, v_A): \exists v_T \text{ with } \mathsf{pc}(v_T, v_S) \text{ iff} \\
& \qquad \text{(i) } \mathsf{token}(v_A) \wedge \forall v_C \text{ s.t. } \mathsf{conn}(v_A, v_C) : \neg\mathsf{offer}(v_C) \text{ or} \\
& \qquad \text{(ii) } \neg\mathsf{token}(v_A) \wedge \exists v_C, v'_A : \mathsf{token}(v'_A) \wedge \mathsf{offer}(v_C) \wedge \\
& \qquad\qquad \mathsf{conn}(v'_A, v_c) \wedge \mathsf{act}(v_C, v_A), \\
& (4)\ \exists v_T, v_S : \mathsf{pc}(v_T, v_S) \iff \neg\exists v'_S : \mathsf{start}(v'_S) \text{ and} \\
& \quad \exists v_A : \mathsf{token}(v_A) \iff \neg\exists v'_A : \mathsf{start}(v'_A) \text{ and} \\
& \quad \neg\exists v_A : \mathsf{start}(v_A) \implies \exists! v'_A : \mathsf{token}(v'_A) \text{ and} \\
& \quad \forall v_C : \mathsf{offer}(v_C) \implies \exists v_A : \mathsf{token}(v_A) \wedge \mathsf{conn}(v_A, v_C) and \\
& \quad \neg\exists v_S : \mathsf{start}(v_S) \implies \exists! v'_S \text{ s.t. } \exists v_T : \mathsf{pc}(v_T, v'_S)
\end{aligned}
$$

We want to show that the relation $\mathcal{R}$ is a weak bisimulation by proving that the states of transition systems can mimic each other moves. The proof uses the propositions and the uniqueness of predicate *corr* (Proposition 5.3.1, exactly one S-node related to one A-node).

Theorem 5.3.5. Given $MT_{\mathcal{A}2\mathcal{B}}$ (as defined in Figure 5.8) and the relation $\mathcal{R}$ (defined in this subsection). Let $G^0_{\mathcal{A}}$, $G^0_{\mathcal{B}}$ be an $\mathcal{A}$- and a $\mathcal{B}$-graph such that $MT_{\mathcal{A}2\mathcal{B}}(G^0_{\mathcal{A}}, G^0_{\mathcal{B}})$. Then the relation $\mathcal{R}$ is a weak bisimulation $\approx$, i.e.

$$map_{\mathcal{A}}(Q(G^0_{\mathcal{A}})) \approx map_{\mathcal{B}}(Q(G^0_{\mathcal{B}}))$$

Proof. of Theorem 5.3.5. Taking the relation $\mathcal{R}$, we need to show the property of mutual simulation. We start with the requirement of initial states being in the relation. The initial states of the LTSs are $G^0_{\mathcal{A}}$ and $G^0_{\mathcal{B}}$ and they satisfy the conditions of $\mathcal{R}$ since they are directly generated by projection from the combined

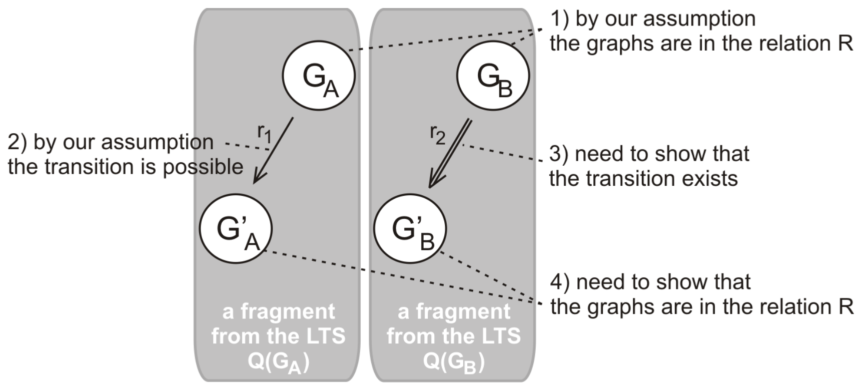

1) by our assumption
the graphs are in the relation R
2) by our assumption
the transition is possible
3) need to show that
the transition exists
4) need to show that
the graphs are in the relation R
G_A
G_B
r1
r2
G'_A
G'_B
a fragment from the LTS Q(G_A)
a fragment from the LTS Q(G_B)

Figure 5.10: The idea of our inductive proof

graph (condition (1)), 5.3.3 guarantees (2) and they have no run-time elements such as tokens, offers or program counters, so condition (3) is trivially satisfied, and (4) follows from the TGG rules.

Now assume $(G_{\mathcal{A}}, G_{\mathcal{B}}) \in \mathcal{R}$ and $G_{\mathcal{A}} \xrightarrow{r_1} G'_{\mathcal{A}}$. As we are looking at the LTSs with labels renamed according to $map_{\mathcal{A}}$ and $map_{\mathcal{B}}$, r_1 (the label of the transition) in principle is either init, move or τ. We need to show that there is some $G'_{\mathcal{B}}$ such that $G_{\mathcal{B}} \overset{\widehat{r_1}}{\Longrightarrow} G'_{\mathcal{B}}$ with $(G'_{\mathcal{A}}, G'_{\mathcal{B}}) \in \mathcal{R}$ (see Figure 5.10). However, as we are interested in the particular semantic rule applied during the step, we will instead directly look at the original LTSs and show that $map_{\mathcal{A}}$ and $map_{\mathcal{B}}$ map rule names to the same label.

$r_1 =$ **initA:** Let $\langle L_1, I_1, R_1, \mathcal{N}_1\rangle$ be the rule for initA in Figure 5.6. If r_1 is applicable in $G_{\mathcal{A}}$, we have a match $m_1 : L_1 \to G_{\mathcal{A}}$, i.e., a node v_S such that start(v_S). From this we construct a match $m_2 : L_2 \to G_{\mathcal{B}}$ for the rule $r_2 =$ initB (both being mapped to init by $map_{\mathcal{A}}$ and $map_{\mathcal{B}}$) being defined as $\langle L_2, I_2, R_2, \mathcal{N}_2\rangle$. The match m_2 maps the A-node in L_2 to the due to Proposition 5.3.2 uniquely existing A-node v_A in $G_{\mathcal{B}}$ such that $corr(v_S, v_A)$. By condition (2) of $\mathcal{R}$ we get start(v_A). Thus r_2 is applicable in $G_{\mathcal{B}}$. Once the rules are applied, we have a graph $G'_{\mathcal{A}}$ with one new T-node v_T with pc(v_T, v_S) minus the (only) start-edge start(v_S), and a similar construction for $G_{\mathcal{B}}$. The pair $(G'_{\mathcal{A}}, G'_{\mathcal{B}})$ is in $\mathcal{R}$ since (1) the syntactic structure without start edges is kept (Proposition 5.3.1); the pair (v_S, v_A) satisfies (2) since both start-edges are deleted, all other pairs satisfy (2) since they are unchanged; (3) is met because we have $\exists v_T$: pc$(v_T, v_S) \land$ token(v_A) and no offers are created, and since by (4) we know that no offers have been existing before; and (4) is met since the two start-edges have been deleted and for them exactly one pc- and one token-edge has been created (and no offers).

$r_1 =$ **movePC:** Since r_1 is applicable in $G_{\mathcal{A}}$, we have nodes v_S, v'_S, v_T in $G_{\mathcal{A}}$ s.t. pc$(v_T, v_S) \land$ next(v_S, v'_S). By (1) and Proposition 5.3.2 there are unique nodes

v_A, v'_A in $G_\mathcal{B}$ s.t. $corr(v_S, v_A)$ and $corr(v'_S, v'_A)$. By (1) and Proposition 5.3.4 there exists v_C s.t. $\mathsf{conn}(v_A, v_C) \wedge \mathsf{act}(v_C, v'_A)$. By (3) there are now two possible cases:

1. $\mathsf{token}(v_A) \wedge \forall v_C$ s.t. $\mathsf{conn}(v_A, v_c) : \neg\mathsf{offer}(v_C)$.
   Thus rule $r_2 = \mathsf{createO}$ matches on v_A and v_C (and both r_1 and r_2 are mapped to move). In the resulting graph $G'_\mathcal{A}$ the pc-edge from v_T to v_S has been deleted and one from v_T to v'_S created. $G'_\mathcal{B}$ has a new offer-edge on v_C. $(G'_\mathcal{A}, G'_\mathcal{B}) \in \mathcal{R}$ since (1) syntactic structure is kept, (2) no start edges are touched, (3) both pairs (v_S, v_A) and (v'_S, v'_A) satisfy the condition, the others are unchanged, and (4) since no start edges are created and the new offer sits on a node following node possessing a token.
2. $\neg\mathsf{token}(v_A) \wedge \exists v_C, v'_A : \mathsf{token}(v'_A) \wedge \mathsf{offer}(v_C) \wedge \mathsf{conn}(v'_A, v_C) \wedge \mathsf{act}(v_C, v_A)$.
   Then the invisible rule moveT (being mapped to τ) is applicable in $G_\mathcal{B}$ leading to a graph $G''_\mathcal{B}$ in which $\mathsf{token}(v_A)$ holds. Moreover, by (4) and rule moveT we know that for all v'_C s.t. $\mathsf{conn}(v_A, v'_C)$ we have $\neg\mathsf{offer}(v'_C)$. Now we reached the first case again and proceed like that. In summary, we get in the renamed LTS

   $$G_\mathcal{B} \xrightarrow{\tau} G''_\mathcal{B} \xrightarrow{\mathsf{move}} G'_\mathcal{B}, \text{ i.e. } G_\mathcal{B} \overset{\widehat{\mathsf{move}}}{\Longrightarrow} G'_\mathcal{B}$$

   and furthermore $(G'_\mathcal{A}, G'_\mathcal{B}) \in \mathcal{R}$.

Reverse direction: assume $G_\mathcal{B} \xrightarrow{r_2} G'_\mathcal{B}$. We need to show that there is some $G'_\mathcal{A}$ such that $G_\mathcal{A} \overset{\widehat{r_2}}{\Longrightarrow} G'_\mathcal{A}$ and $(G'_\mathcal{A}, G'_\mathcal{B}) \in \mathcal{R}$. Again, we argue on the level of LTSs before renaming.

$r_2 = \mathsf{initB}$**:** Similar to initA.

$r_2 = \mathsf{createO}$**:** Since r_2 is applicable in $G_\mathcal{B}$ there are nodes $v_A, v_C : \mathsf{token}(v_A) \wedge \mathsf{conn}(v_A, v_C) \wedge \forall v'_C$ s.t. $\mathsf{conn}(v_A, v'_C) : \neg\mathsf{offer}(v_C)$. By (1) and Proposition 5.3.2 there is a unique node v_S s.t. $corr(v_S, v_A)$. By (3) $\exists v_T : \mathsf{pc}(v_T, v_S)$. By (1) and Proposition 5.3.4 $\exists v'_A, v'_S$ s.t. $\mathsf{next}(v_S, v'_S) \wedge \mathsf{act}(v_C, v'_A) \wedge corr(v'_S, v'_A)$. Hence rule movePC (mapped to move like $\mathsf{createO}$) is applicable in G_A. The rest follows from a reasoning similar to case movePC.

$r_2 = \mathsf{moveT}$**:** In this case, we have an invisible step on the $\mathcal{B}$-side. If r_2 is applicable in $G_\mathcal{B}$, then $\exists v_A, v_C, v'_A : \mathsf{token}(v_A) \wedge \mathsf{offer}(v_c) \wedge \mathsf{conn}(v_A, v_C) \wedge \mathsf{act}(v_C, v'_A)$. By (1) and Proposition 5.3.2 $\exists v_S, v'_S : corr(v_S, v_A) \wedge corr(v'_S, v'_A)$. By Proposition 5.3.2 and 5.3.4 we get $\mathsf{next}(v_S, v'_S)$. By (4) we get $\neg\mathsf{token}(v'_A)$. By (3) we have $\exists v_T : \mathsf{pc}(v_T, v'_S)$ and thus by (4) $\neg\exists v_T : \mathsf{pc}(v_T, v_S)$. Applying rule r_2 leads to a graph $G'_\mathcal{B}$ in which $\mathsf{token}(v'_A)$ and $\neg\mathsf{offer}(v_C) \wedge \neg\mathsf{token}(v_A)$ holds. Because of (4) (the only possible offer was on v'_C) we know that $\forall v'_C$ s.t. $\mathsf{conn}(v'_A, v'_C) : \neg\mathsf{offer}(v'_C)$. The pair $(G_\mathcal{A}, G'_\mathcal{B})$ is thus in $\mathcal{R}$ and furthermore $G_\mathcal{A} \overset{\widehat{\tau}}{\Longrightarrow} G_\mathcal{A}$ which completes the proof.

□

5.3.6 Summary

In this section we performed a five step method, which allows to assure the correctness of model transformation in the sense of behavioural preservation. The method was illustrated on two self-defined simple languages. Figure 5.11 summarizes this section, by presenting the sequence of steps we made. Firstly, we defined the syntax of two languages with type graphs. In the next step, we defined semantics of two languages by means of graph transformations. Thirdly, we mapped the rule systems to a single universe. Fourthly, we specified model transformation for the languages with TGG rules. Finally, we showed how to prove the correctness of the model transformation by establishing a weak bisimulation over the states of LTSs: the first LTS is generated for a source graph, by application of graph transformation rules that define semantics for the source language, and the second LTS is generated for a target graph, which is a result of the model transformation.

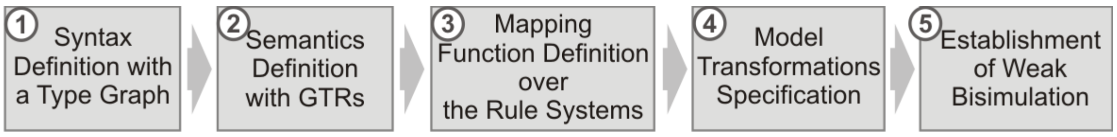

1 Syntax Definition with a Type Graph
2 Semantics Definition with GTRs
3 Mapping Function Definition over the Rule Systems
4 Model Transformations Specification
5 Establishment of Weak Bisimulation

Figure 5.11: Summary of the method

5.4 Interpretation of Behavioural Properties

The presented method shows (1) how to specify a model transformation $MT_{\mathcal{A}2\mathcal{B}}(G_{\mathcal{A}}, G_{\mathcal{B}})$, which transforms a graph $G_{\mathcal{A}}$ into a graph $G_{\mathcal{B}}$, (2) how to define a relation over the well-formed run-time graphs $\mathcal{R} \subseteq \mathcal{G}^{\mathsf{rt}}_{\mathcal{A}} \times \mathcal{G}^{\mathsf{rt}}_{\mathcal{B}}$, which form the states in LTSs $Q(G_{\mathcal{A}})$ and $Q(G_{\mathcal{B}})$, (3) how to prove that the defined relation is a weak bisimulation, i.e.

$$Q(G_{\mathcal{A}}) \quad \approx \quad Q(G_{\mathcal{B}})$$

From the previous chapter, we conclude that weak bisimulation on LTSs implies the preservation of behavioural properties, i.e.

$$Q(G_{\mathcal{A}}) \models \varphi \quad \Rightarrow \quad Q(G_{\mathcal{B}}) \models \chi(\varphi)$$

here, φ is an ACTL formula specified over the labels of transitions in an LTS with $AP = dom(\mathcal{RS}_{\mathcal{A}})$. The function χ is an interpretation of φ over the target language, i.e. over an LTS with $AP = dom(\mathcal{RS}_{\mathcal{B}})$.

Recall that the TGG rules $MT_{\mathcal{A}2\mathcal{B}}$ specify how to transform syntactic structure of a graph $G_{\mathcal{A}}$ into a graph $G_{\mathcal{B}}$. The relation $\mathcal{R}$ uses the correspondences, which are generated by MT_{A2B}, to specify (1) conditions on a syntactic structure of the graphs during the semantic execution and (2) all possible pairs of the source and target graph structures, which exhibit run-time properties on corresponding nodes. The goal of this section is to specify the interpretation function $\chi : \Phi_{\mathcal{A}} \mapsto \Phi_{\mathcal{B}}$, where $\Phi_{\mathcal{A}}$ is a set of all behavioural properties $\varphi_{\mathcal{A}}$ of a model (specified as a graph $G_{\mathcal{A}}$), $\Phi_{\mathcal{B}}$ is a set of all behavioural properties $\varphi_{\mathcal{B}}$ of a model (specified as a graph $G_{\mathcal{A}}$). In the follow, we explain a scenario of behavioural properties interpretation (or the meaning of function χ).

A behavioural property is specified with an ACTL formula, where the set of atomic propositions is an extended rule system $\mathcal{RS}^{+}$, as explained in Section 4.4. To interpret a behavioural property of the source model, we use the weak bisimulation $\mathcal{R}$ as an interlink, because it is defined over the well-formed run-time graphs and specifies the connection of all the possible run-time instances (see the diagram below).

$$\begin{tikzcd}
 & \mathcal{G}^{\mathsf{rt}}_{\mathcal{A}} & \mathcal{R} \arrow[l] \arrow[r] & \mathcal{G}^{\mathsf{rt}}_{\mathcal{B}} & \\
 & dom(\mathcal{RS}^{+}_{\mathcal{A}}) \arrow[u] & \overset{?}{\to} & dom(\mathcal{RS}^{+}_{\mathcal{B}}) \arrow[u] & \\
Property \arrow[r, Rightarrow] & \varphi \arrow[u] & \overset{?}{\to} & \chi(\varphi) \arrow[u] & Property \arrow[l, Rightarrow]
\end{tikzcd}$$

Here, the (standard type) arrows denote the relation "specified over". The main question is how to interpret the set of atomic propositions and an ACTL formula for the target language.

The process of properties interpretation is described as follows:

(1) At first, a behavioural property must be specified according to Section 4.4. For this, we either (a) use one or more transformation rules $r_{\mathcal{A}}$ from the graph transformation rule system $\mathcal{RS}_{\mathcal{A}}$ that specifies the behaviour of the source language, or (b) specify one or more graph transformation rules $r_{\mathcal{A}}$, where the left-hand and a right-hand sides coincide. In the case (b), the rule system is extended with $r_{\mathcal{A}}$ and is denoted by $\mathcal{RS}^{+}_{\mathcal{A}}$. Then, a behavioural property can be specified as an ACTL formula over $AP = dom(\mathcal{RS}^{+}_{\mathcal{A}})$.

(2) We consider further two cases introduced earlier. Case (a), when $r_{A} \in \mathcal{RS}_{\mathcal{A}}$. Then in order to interpret this case for the target $\mathcal{B}$-language, it is required to use the mappings $map_{\mathcal{A}}$ and $map_{\mathcal{B}}$. These mappings were defined over the graph transformation rules in order to map the rules from $\mathcal{RS}_{\mathcal{A}}$ and $\mathcal{RS}_{\mathcal{B}}$, which perform similar behaviour (see Step 3 of our method). The rule $r_{\mathcal{A}}$ must be mapped to one or more graph transformation rules $r_{\mathcal{B}}$ from the rule system $\mathcal{RS}_{\mathcal{B}}$. If the rule $r_{\mathcal{A}}$ is mapped to an invisible step, then the property it specifies

can not be interpreted into another language. Otherwise, continue with step (5).

Now consider case (b), when $r_{\mathcal{A}} \notin \mathcal{RS}_{\mathcal{A}}$. Then the rule $r_{\mathcal{A}}$ consists of one or more nodes and specifies the conditions on a graph structure that could be also written down as a logical statement $cond_{\mathcal{A}}$.

(3) $\mathcal{R}$ is defined as a set of conditions on the run-time structures. Since the rule $r_{\mathcal{A}}$ can be considered the condition $cond_{\mathcal{A}}$ which describes a property of a source run-time structure, the interpretation of the rule $r_{\mathcal{A}}$ is done by analysis of the correspondences in the TGG graph and the conditions of $\mathcal{R}$. For example, if the rule $r_{\mathcal{A}}$ states that there must be a connection between S-node and T-node, then there exists a corresponding A-node. The condition (3) in the relation $\mathcal{R}$ defines two cases for the corresponding S- and A-nodes. There are two options for the corresponding node v_A (when the corresponding S-node is connected with a T-node):

(i) $\mathsf{token}(v_A) \wedge \forall v_C$ s.t. $\mathsf{conn}(v_A, v_C) : \neg\mathsf{offer}(v_C)$ or

(ii) $\neg\mathsf{token}(v_A) \wedge \exists v_C, v'_A : \mathsf{token}(v'_A) \wedge \mathsf{offer}(v_C) \wedge \mathsf{conn}(v'_A, v_c) \wedge \mathsf{act}(v_C, v_A)$,

These two options can be formalized as a graph structure, which was already demonstrated in Figure 5.9. By restriction of such structure on the target meta-model, we receive two graph patterns $r^1_{\mathcal{B}}$ and $r^2_{\mathcal{B}}$, which are the interpretation of $r_{\mathcal{A}}$ in the target $\mathcal{B}$-language.

(4) The rule system $\mathcal{RS}_{\mathcal{B}}$ is extended with one ore more graph transformation rules $r_{\mathcal{B}}$, where left-hand and right-hand sides coincide. In the example used above, the rule system $\mathcal{RS}_{\mathcal{B}}$ is extended with the graphs $r^1_{\mathcal{B}}$ and $r^2_{\mathcal{B}}$.

(5) The formula φ is interpreted for the LTS $Q(G_{\mathcal{B}})$ over the new set of atomic propositions $AP = dom(\mathcal{RS}^+_{\mathcal{B}})$. In case the rule $r_{\mathcal{A}}$ has two or more corresponding graph structures, it is replaced in an ACTL formula with $r^1_{\mathcal{B}} \vee r^2_{\mathcal{B}} \vee \ldots$.

We illustrate the interpretation of properties during model transformation by performing an example, where we consider a concrete graph of the $\mathcal{A}$-language, which is transformed into a graph of the $\mathcal{B}$-language. We specify a sample liveness property and a sample safety property for the $\mathcal{A}$-language, then we describe how the properties are interpreted for the transformed $\mathcal{B}$-language to demonstrate the results.

Example 5.4.1 (Properties interpretation)**.** *We consider a TGG graph $G_{\mathcal{AB}}$ which is a product of the TGG rules from Figure 5.8. The graph is depicted in Figure 5.12. The graph $G_{\mathcal{AB}}$ consists of three subgraphs, two of which are the graph $G_{\mathcal{A}}$ and $G_{\mathcal{B}}$ typed over the type graphs $T^{st}_{\mathcal{A}}$ and $T^{st}_{\mathcal{B}}$, respectively. The graph $G_{\mathcal{A}}$ (see Figure 5.4 in the left) is a source graph and the graph $G_{\mathcal{B}}$ (see Figure 5.4 in the right) is a target graph. The corresponding graph, which consists of* G*-nodes, connects the* S*-nodes and the* A*-nodes in $G_{\mathcal{AB}}$.*

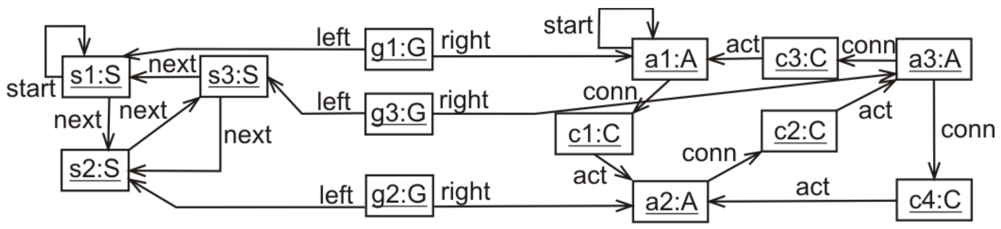


Figure 5.12: The TGG graph $G_{\mathcal{AB}}$

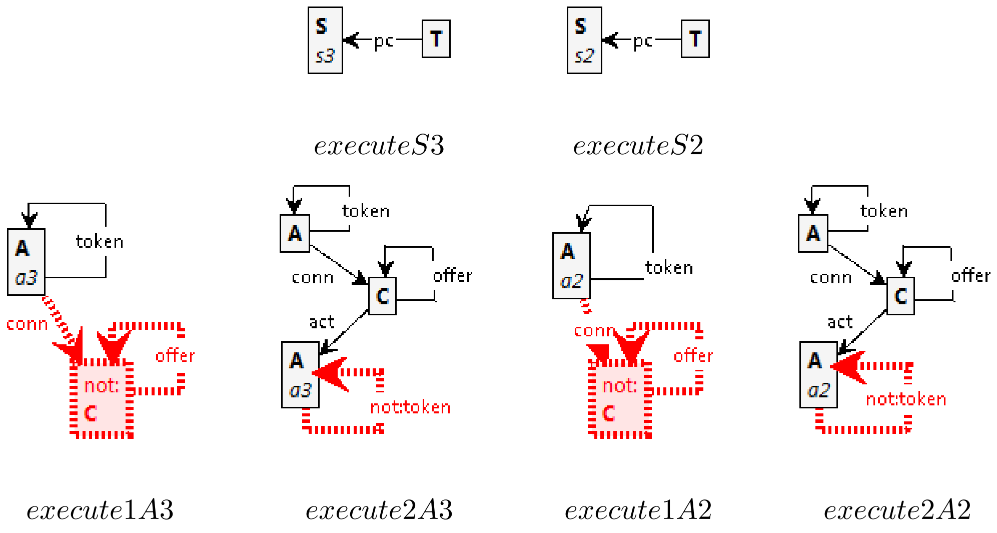


Figure 5.13: Graph transformation rules which extend the original rule systems

We already know that the LTSs generated by the semantic rules being applied to the graphs $G_{\mathcal{A}}$ and $G_{\mathcal{B}}$ are weak bisimilar. Now we want to verify concrete properties of the source language $\mathcal{A}$, to interpret it for the language $\mathcal{B}$ and show that they hold.

One property is a liveness property specifying that the state representing the pc*-edge flowing through the node s3 must be reachable infinitely often. To write it down formally, we do some preparatory steps (see Section 4.4 for more details). At first, we notice that no rule from $\mathcal{RS}_{\mathcal{A}}$ allows to specify the flow of dynamic elements throw particular nodes. Therefore, we enrich the rule system $\mathcal{RS}_{\mathcal{A}}$ with an additional rule $executeS3$, that does not change a graph during the semantics execution. The rule $executeS3$ requires the existence of a* S*-node s3 and the* T*-node, which is connected with s3 by a* pc*-edge (see Figure 5.13 in the top).*

The second property we want to verify is that the nodes s2 and s3 are never connected with the pc*-edge simultaneously. The definition of the property $executeS2$*

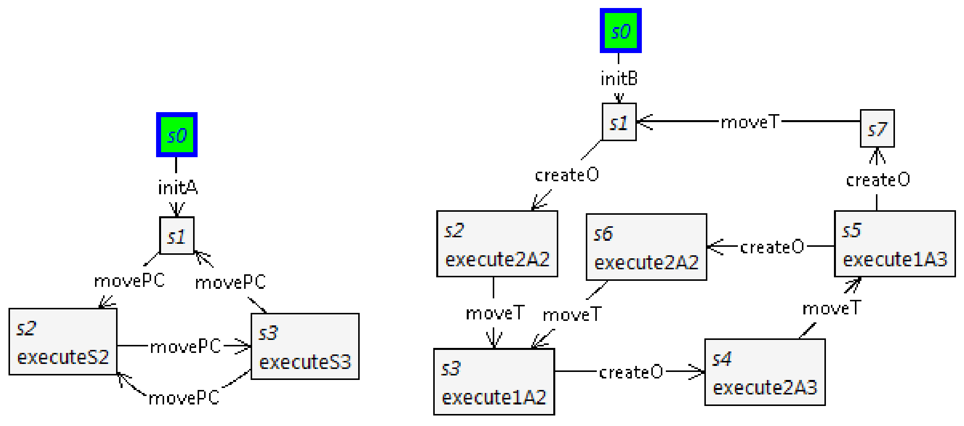


Figure 5.14: The LTS for $G_{\mathcal{A}}$ in the left and the LTS for $G_{\mathcal{B}}$ in the right

is similar to the definition of the property $executeS3$ *(see the result on Figure 5.13 in the top). The property* $executeS2$ *is a safety property.*

The LTS for the graph $G_{\mathcal{A}}$ *generated by the extended rule system* $\mathcal{RS}^{+}_{\mathcal{A}}$*, where* $dom(\mathcal{RS}^{+}_{\mathcal{A}}) = dom(\mathcal{RS}_{\mathcal{A}}) \cup \{executeS2, executeS3\}$*, is presented in Figure 5.14 in the left. There are four states. The transitions with the labels* $executeS2$ *and* $executeS3$ *are generated by the graph transformation rules* $executeS2$ *and* $executeS3$ *which play the role of invariant. The transitions do not modify the graph, therefore they are loop transitions.*

We specify the properties by means of ACTL:

$$\varphi_1 = \forall G(\exists F(executeS3))$$

$$\varphi_2 = \forall G \neg(executeS3 \wedge executeS2)$$

Formula φ_1 *holds in every state, as in any state of any of its paths it is possible to reach the state* s_3*. Formula* φ_2 *is valid for all states too, because for any path from each state there is no state, where* $executeS3$ *and* $executeS2$ *are both globally valid.*

In the next step we interpret the defined properties for the $\mathcal{B}$*-language. We start with the property* φ_1*. It specifies the conditions on the graph structure, which could be written down as follows:* $\exists v_S = s_3 : pc(v_T, s_3)$*. The node* $a3$ *is a corresponding node for* $s3$*, i.e.* $\mathsf{conn}(s3, a3)$*. According to the conditions for the relation* $\mathcal{R}$ *(see Section 5.3.3), there are two conditions for the node* $a3$*. The first one is that* $a3$ *must have a self edge of* token*-type and there must be a* C*-node, such that the* C*-node is connected to* $a3$ *with a* conn*-edge and it has a self edge of* offer*-type (see Figure*

5.13). The second one is that if there exists a A*-node and a* C*-node, such that there is a* conn*-node between them and there exists an* act*-node between the* C*-node and* $a3$*, then the* A*-node and the* C*-node must have self edges of* token*-type and* offer*-type, respectively (see Figure 5.13 in the right). We extend the rule system* $\mathcal{RS}_{\mathcal{B}}$ *with two graph transformation rules* $execute1A3$ *and* $execute2A3$*, which do not change the graph and assure that the described conditions hold on a certain step of semantic execution.*

To interpret the property φ_2 *we examine again the relation* $\mathcal{R}$ *and the corresponding node* $a2$*. Similar to the property* φ_1*, we extend the rule system* $\mathcal{RS}_{\mathcal{B}}$ *with two graph transformation rules* $execute1A2$ *and* $execute2A2$ *(see Figure 5.13). The interpreted properties with respect to the mapping are the following:*

$$\chi(\varphi_1) = \forall G(\exists F(execute1A3 \vee execute2A3))$$

$$\chi(\varphi_2) = \forall G\neg((execute1A2 \vee execute2A2) \wedge (execute1A3 \vee execute2A3))$$

Further, we generate an LTS for the graph $G_{\mathcal{B}}$ *(see the results in Figure 5.14 in the right). The LTS* $(Q(G_{\mathcal{B}}))$ *satisfies both properties* $\chi(\varphi_1)$ *and* $\chi(\varphi_2)$*.*

5.5 Summary

In this chapter we presented a method which establishes a weak bisimulation over the states of the LTSs, generated for *any* source graph and target graph, where the later is a result from a model transformation. The method was applied to a concrete example of two toy graph languages, $\mathcal{A}$ and $\mathcal{B}$. It was shown that for any graph $G_{\mathcal{A}}$ of $\mathcal{A}$-language and a graph $G_{\mathcal{B}}$ of $\mathcal{B}$-language resulting from its transformations, the LTSs are weak bisimilar with respect to the mapping of the behavioural semantics rules. We explained how to interpret the method results, i.e. the behavioural properties preservation during a model transformation.

CHAPTER 6

Case Study: Model Transformation of CCS into Petri Nets

In this chapter we provide a case study in order to show that the method presented in the previous chapter is applicable to real languages. The idea of the case study is to specify a model transformation between two languages, Calculus of Communication Systems (CCS) [Mil95] and Petri nets [Rei85] (see Figure 6.1). We use our method (see Figure 5.11) to specify the languages and the model transformation by means of graph transformations and to prove the behaviour preservation during the model transformation.

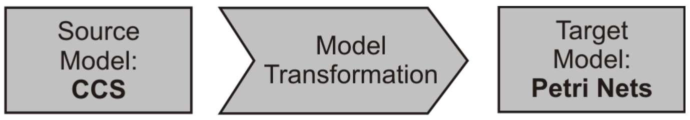


Figure 6.1: General idea of this chapter: model transformation between CCS and Petri nets

There is a number of reasons for this particular choice of languages for the case study. Firstly, we decided to apply our method to two real languages, between which the behaviour is already studied. This allows us to study the applicability of our method. The connection between the CCS language and Petri nets has been studied earlier [Old86, CMPS82] and the weak bisimulation relation for the languages was already proven [Gol88]. Another important reason for our choice is that the languages are comparatively less complicated than modelling and programming languages, such as UML or Java [EHSW99]. Thirdly, the languages are equipped with a formal standard semantics.

The chosen languages fit to the MDA idea about vertical model transformation (see Chapter 1), which are done with the purpose of improving the quality of models at a particular level of abstraction. Both languages are used for model specification on a platform independent level. The CCS language is used for modelling concurrency, and Petri nets are widely used to simulate nondeterministic computation. With our method we show that it is possible to define a model transformation between these two languages by means of graph transformations and show that this transformation preserves behavioural properties.

Our transformation is mainly based on [GM84, Gol88], where the authors consider a restricted CCS language. The first restriction is related to the fact that the CCS language is Turing powerful and finite Petri nets are not. The second restriction is related to a problem of implementing a choice operator in Petri nets. We show that the second restriction is not needed, by implementing the behaviour semantics by means of graph transformations. Similar to [Gol88] we are interested in a transformation of CCS into finite Petri nets. In opposite to [Gol88] we do not deviate from a usual definition of Perti nets.

The model transformation between CCS and Petri nets involve several stages (see Figure 6.2). Beside the model transformations (Stage 3), the syntax of both models needs to be defined with a type graph and the formal semantics must be specified with graph transformations in order that our method could be applied. For this we perform Stage 1, when the original syntax of CCS defined with extended Backus-Naur Form (EBNF) [Ove97] is transformed into meta-model notation and then the meta-model is transformed into a Type Graph (TG). The CCS semantics originally defined by the Interleaving Operational Semantics (IOS) [DNM88] is also transformed into a semantics based on Graph Transformation Rules (GTRs). We prove that the new notation has the same behavioural semantics by comparison of LTSs.

The original definition of the Petri net language is not provided by means of graph transformations, but within Set Theory (ST) and the behavioural semantics is defined with a usage of a function, called *Marking* (M). However, there are examples in the literature [BEMS08, MEE10], where the Petri nets language is defined with graphs and graph transformations and there is a standard graphical representation of Petri nets, which is very similar to a graph representation. Therefore, we do not present a proof of behaviour preservation similar to CCS, instead we introduce the Petri nets language by means of graph transformations with formal explanations of implementing the original specification (Stage 2).

So, there are three stages in this chapter. The first stage is the definition of syntax and semantics of CCS with graph transformations. The second stage is representing the Petri nets language by means of graph transformations. The third

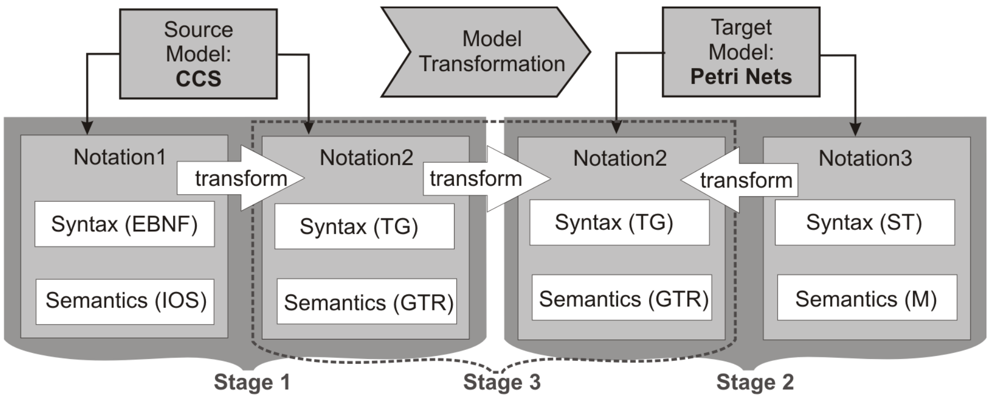


Figure 6.2: Several stages of transformations: the first stage of transformations - the original syntax of CCS defined with EBNF is transformed into a TG and the IOS into the semantics based on GTRs, the second stage of transformations – the original syntax of Petri nets defined with ST is interpreted as a TG and the semantics based on a usage of a Marking (M) function is implemented with GTRs, the third stage of transformations is a model transformation, which is represented within the same notation

stage is the model transformation between the CCS language and the Petri nets language. The semantic preservation is proven in first and third stages.

We proceed as follows. This chapter is organized as a case study of our five step method (Figure 6.3). Sections 6.1 and 6.2 include Steps 1-2 of our method. The former section covers the earlier described Stage 1. The latter section specifies the Petri net language by means of graph transformations, i.e. Stage 2. Furthermore, we proceed with Stage 3. We provide a step 3 of our method: mapping of graph transformation rules from rule systems which define the semantics of the languages (Section 6.3). The transformations from CCS into Petri nets (Step 4) are presented in Section 6.4. In Section 6.5 we show Step 5: the behavioural preservation during model transformations. We conclude this chapter by an illustration of the results of our method by verification of some properties on a concrete example in Section 6.6.

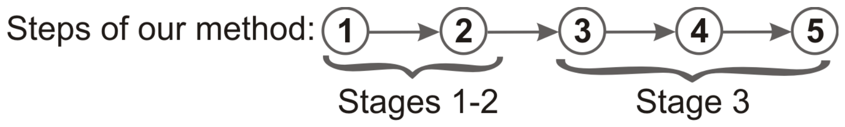


Figure 6.3: Connection between stages and steps of the method

6.1 CCS Language (Steps 1-2)

In this section, we provide Steps 1-2 of our method, i.e. the definition of the syntax and the semantics of the CCS language by means of graph transformations. Since the original definition is different, we provide a step-wise approach, or we also say that we perform substeps. As a first substep (see Figure 6.4), we introduce the CCS original syntax defined with the EBNF and the CCS semantics given in the IOS style. In the second substep, we transform the EBNF into a type graph. Then, we extend a type graph to a run-time graph and we define the graph transformation rules over the run-time graph. Finally, we prove that the newly defined graph transformations describe the same behaviour as the original IOS. Therefore, there are intermediate substeps in this section, beside the definition of the CCS language with graph transformations.

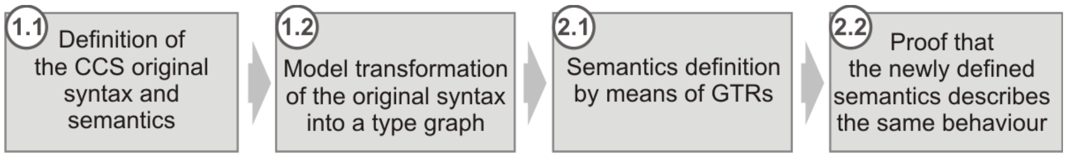

1.1
Definition of the CCS original syntax and semantics
1.2
Model transformation of the original syntax into a type graph
2.1
Semantics definition by means of GTRs
2.2
Proof that the newly defined semantics describes the same behaviour

Figure 6.4: Overview of this section

6.1.1 Original Syntax and Semantics

We start with a small example. Let us consider a system: a banking system webpage (see the interface of a webpage in Figure 6.5). We suppose that a visitor can login to online-banking, for this he needs to provide a username and a password, we denote these *events* as c and d, respectively. The visitor can also find the nearest office by providing an address and pressing the button “Find our offices”, we denote these *events* as a and b, respectively. One natural way to define the banking system webpage, as a *CCS process* P, is in terms of its interaction with the environment at its four ports (a, b, c, d), as follows:

$$P = a.b.nil + c.nil \mid d.nil$$

This means that you can either, for example, login in a system, for this you must provide a username and a password (what to provide firstly is not specified), or you can find an office, then you need to provide an address and then to press the button “Find our offices”. Note that the system behaviour will not let you do both operations, you need to decide either you want to proceed with online-banking or you want to know the office.

In our example, we illustrated an event (e.g. a), a sequence of events ($a.b.nil$), a choice of events ($a.nil + b.nil$), events executed in parallel ($c.nil \mid d.nil$). Beside

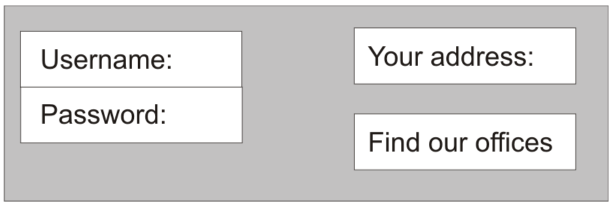


Figure 6.5: An example of a web page interface

these operators, there is also a recursion ($\mu x.a.x$). We do not consider relabelling and restriction [Mil95] and therefore restrict the original CCS language. Now it will be easier to understand the syntax of the CCS language. It is originally given in terms of grammar rules. Below is the EBNF grammar of the CCS language:

$$\begin{array}{l} <\!P\!> \;::= nil \;\mid\; <\!V\!> \;\mid <\!E\!>.<\!P\!> \;\mid\; <\!P\!> + <\!P\!> \;\mid \\ \;\; <\!P\!> \parallel <\!P\!> \;\mid\; \mu <\!V\!>.<\!P\!> \\ <\!E\!> \;::=\; a \mid b \mid c \ldots \\ <\!V\!> \;::=\; x \mid y \mid z \ldots \end{array} \qquad (1)$$

Here E stands for the action name from the set Act, with the following structure: $Act := \Lambda \cup \{\tau\}$; $\Lambda := \Delta \cup \bar{\Delta}$, where Δ is a set of names, $\bar{\Delta} := \{\bar{a} | a \in \Delta\}$ and the mapping $a \mapsto \bar{a}$ is a bijection. We call a an action and $\bar{a}$ a co-action. V is a set of variables, which is used for the recursion.

The intuitive meaning of the non-terminals is as follows. *nil* is not able to perform any action, $a. < P >$ performs a and then behaves like $< P >$. It also means that the event a occurs. In $< P > \parallel < P >$ (later denoted as $< P > | < P >$, note that the processes could be different), the processes $< P >$ are executed concurrently; complementary actions may be performed jointly as a τ-action. $< P > + < P >$ behaves like one of the processes $< P >$. Operators $+$ and $|$ are associative and communicative, therefore the order of components is irrelevant. $\mu x. < P >$ declares a variable x, a recursive invocation happens, when x occurs in $< P >$ without μ. All mentioned operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right. The order for all operators from higher priority to low priority is: (), |, +, μ and ., where the last two operators have the same priority.

To define a notion of behavioural semantics, we use the notion of Labelled Transition System (LTS) defined in Chapter 4. Thus, we have a structure $Q = \langle S, \rightarrow, \iota, Act \rangle$, which stands for LTS. Here, S is a set of states. In our case the states are the CCS processes, i.e. the words of the CCS language. $\xrightarrow{a} \subseteq S \times Act \times S$ is a transition relation, where each transition $\rightarrow$ is labelled over the set Act. ι is a set of initial states. In IOS there exists an LTS for each process P.

The CCS behavioural semantics consists of the definition of each transition relation $\xrightarrow{a}$ over *Act*. Further, we explain how a transition $\rightarrow$ is defined. For this, we introduce a set of inference rules, which are used for *justification* of a transition, i.e. the transition could be inferred by the set of rules. Then, an LTS for a process P is defined by all justified transitions.

Prefixing $a.P \xrightarrow{a} P$

CompositionI $\dfrac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q}$ $\quad \dfrac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'}$

CompositionII $\dfrac{P \xrightarrow{a} P' Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$

Summation $\dfrac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'}$ $\quad \dfrac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'}$

Recursion $\dfrac{P[\mu x P/x] \xrightarrow{a} P'}{\mu x P \xrightarrow{a} P'}$

where $P[\mu x P/x]$ denotes the term resulting from the substitution of all free occurrences of x in P by $\mu x P$.

We say that our set of rules is complete; by this we mean that there are no transitions except those which can be inferred or deduced by the rules.

We return to our example, we want to illustrate how an LTS could be received for a process $P = a.b.nil + c.nil \mid d.nil$. For this, we consider a transition.

$$a.b.nil + c.nil \mid d.nil \xrightarrow{d} c.nil \mid nil$$

We need to show or *justify* that this transition could be inferred by the previously defined rules. We can now set out the justification for a transition of a CCS process in the form of an inference diagram, in which we annotate each inference with the name of the IOS rule which justifies it. The justification is given as follows:

$$\dfrac{\dfrac{\dfrac{}{d.nil \xrightarrow{d} nil}\ \text{Prefixing}}{c.nil \mid d.nil \xrightarrow{d} c.nil \mid nil}\ \text{CompositionI}}{a.b.nil + c.nil \mid d.nil \xrightarrow{d} c.nil \mid nil}\ \text{Summation}$$

At first, we applied the Summation rule, then the CompositionI rule and the Prefixing rule. If we unify all possible transitions for the process P (we omit the justification process for each single transition), then we get the LTS as it is shown in Figure 6.6.

6.1.2 From EBNF to Meta-Model

In this subsection, we want to define the CCS syntax with a type graph. For this we firstly transform the EBNF form into a meta-model. Then, we define a type

$$\begin{tikzcd}
& P = a.b.nil + c.nil|d.nil \arrow[dl, "a"'] \arrow[d, "d"'] \arrow[dr, "c"] & \\
b.nil \arrow[d, "b"'] & nil|c.nil \arrow[d, "c"'] & d.nil|nil \arrow[d, "d"'] \\
nil & nil|nil & nil|nil
\end{tikzcd}$$

Figure 6.6: Transition system generated by IOS for the process $P = a.b.nil + c.nil \mid d.nil$

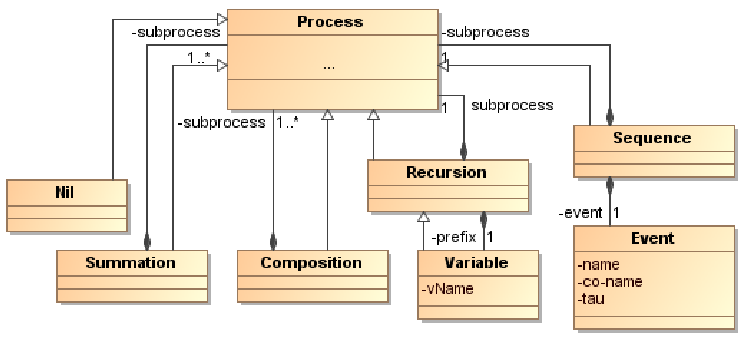


Figure 6.7: Meta-model for the syntax of the CCS language

graph for the received meta-model.

We use the research [WK05] to transform the EBNF grammar of the CCS language into a meta-model. The purpose of this research is to transform the textual definitions of EBNF into a format, that can be processed by model engineering tools. The transformation is based on the following rules. The left-hand side of a production rule is transformed to a class. The elements of the right-hand side are connected to the left-hand side class by a containment association with role-names to indicate the subprocesses, events and variables and contain the constraints. To simplify the instances of meta-model, we include the inheritance relation between the classes which correspond to a process P and the classes which correspond to the right-hand side of a first production rule. Non-terminals turn into attributes. For simplicity reasons, we define three different attributes for events from set $Act = \Delta \cup \bar{\Delta} \cup \{\tau\}$, each subset of Act has a unique attribute. See the result in Figure 6.7.

In our research we do not use the meta-model directly, rather we encode it into a graph structure. Class and object diagrams can be treated as (labelled) graphs (see Chapter 3). Therefore, we use later type graph T^{st}_{CCS} (see Figure 6.8), received by direct translation of the meta-model.

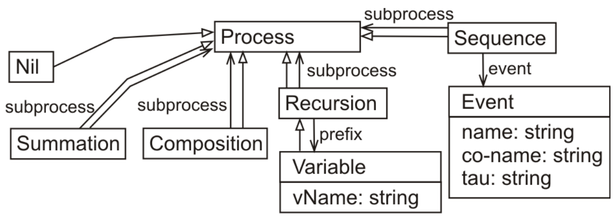


Figure 6.8: Type graph T^{st}_{CSS}

We want to take into account the constraints on the associations during the transformation of meta-model into a type graph. For this we impose the constraints on the syntactic graph structure. The type graph must impose only a weak structure: not all graphs that can be typed over T^{st}_{CSS} are considered to be parts of the language. We define the constraints by specifying the well-formed graphs. Then, the CCS language defined by the type graph T^{st}_{CSS} consists of only well-formed graphs. Well-formedness is inductively defined (similar approaches to well-formedness can be found in [PKT73, EKR$^+$08]). For this, we introduce the concept of *building blocks*. Every building block is a Process-node (see Figure 6.9 in the top left corner) and corresponds to the first left-hand side of a production rule from EBNF grammar (see Subsection 6.1.1). For the right-hand side of a production rule we define the following patterns:

- A Nil-node itself constitutes a building block (see Figure 6.9A) and corresponds to a process *nil*.
- A Summation-node, followed by two Process-nodes, is a building block (see Figure 6.9B) and corresponds to a process $P_1 + P_2$ [1].
- A Composition-node, followed by two Process-nodes, is a building block (see Figure 6.9C).
- A Sequence-node, followed by an Event-node and a Process-node, is a building block (see Figure 6.9D) and corresponds to a process $E.P$. An event $a \in Act$ corresponds to an attribute value of the Event-node.
- A Recursion-node, followed by a Variable-node and a Process-node, is a building block (see Figure 6.9E) and corresponds to a process $\mu V.P$.
- A Variable-node itself constitutes a building block (see Figure 6.9F) and corresponds to a process V.

An example of a well-formed typed graph over T^{st}_{CSS} is depicted in Figure 6.10. Since each well-formed construction is defined by EBNF, then we can map each process to a well-formed CCS graph. We say later that there is a *corresponding* graph G_P for the process P. A corresponding graph for the process $P[\mu xP/x]$, which means that all occurrences of a variable x in P are replaced with P (a part

[1]Instead of writing $< P_1 > + < P_2 >$ we skip the parenthesis and write $P_1 + P_2$ further.

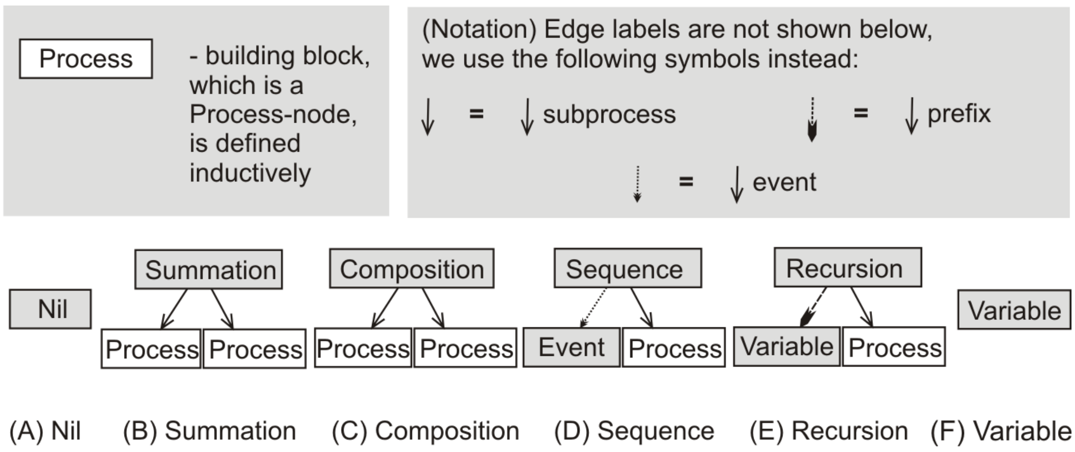


Figure 6.9: Definition of well-formed CCS graphs

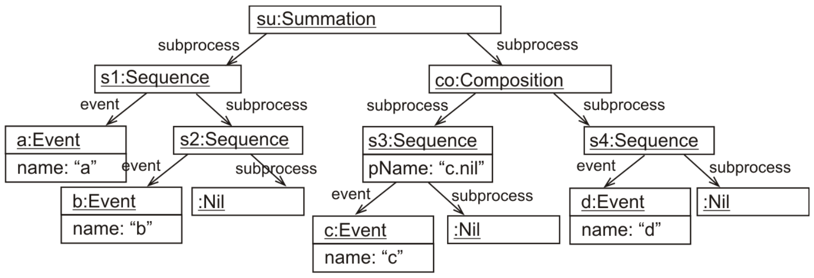


Figure 6.10: The CCS graph for the process $P = a.b.nil + c.nil \mid d.nil$

of the inference rule Recursion), is a graph G_P, where each structure is mapped as it is defined by a well-formed construction.

Note that a CCS typed graph represents always a tree, the intermediate nodes of which are Process-nodes; Event-nodes, Nil-nodes and Variable-nodes are leaves. All nodes that inherit the Process-nodes are the roots of some subtrees in the instance model. These subtrees correspond to *subprocesses* of the process P (subprocess is defined below).

Definition 25 (Subprocess)**.** Let P be a CCS process, then P' is a *subprocess* of P (denoted later also as $P \succ P'$), if

(a) P' is defined by EBNF grammar (1), and

(b) P is defined by EBNF grammar (1), where at least one of the components (i.e. $< P >$) is P'.

Note that each process x is always a subprocess of some process $\mu x.P$. Similarly, a graph G_x (which is a Variable-node) is always a leaf node of a graph $G_{\mu x.P}$.

We use $\mathcal{G}^{\mathsf{st}}_{CCS}$ to denote the set of all well-formed (syntactic) CCS graphs.

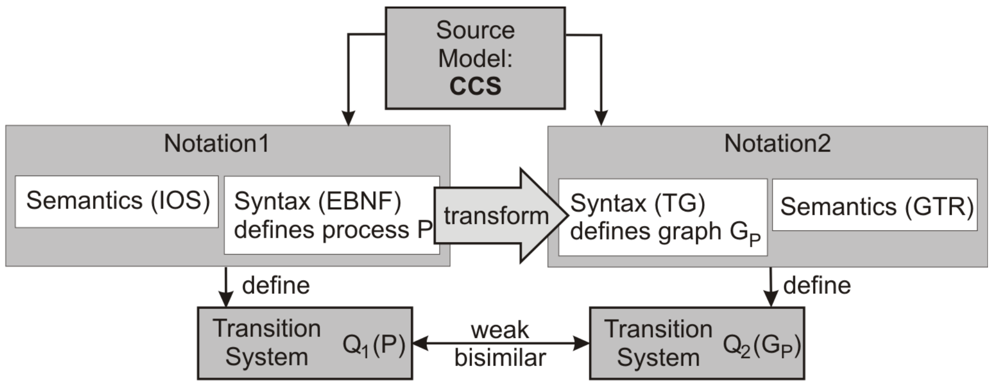


Figure 6.11: The idea to implement behavioural semantics by means of graph transformations: a transition system generated for the original CCS process must be weak bisimilar with a transition system generated for the corresponding graph

6.1.3 From Interleaving Operational Semantics to Semantics Defined by Graph Transformations

After we defined the CCS syntax with a type graph, we continue with the definition of the CCS semantics by means of graph transformations. Later in Subsection 6.1.4 we prove that our definition of behavioural semantics is correct. The content of this and the next subsections is Step 2 of our method.

Let Q_1 be an LTS generated for a process P by IOS semantics (denoted later as $Q_1(P)$). Let G_P be a corresponding graph for process P. An LTS generated for G_P with graph transformation rules (defined later for the behavioural semantics of the CCS language) is denoted as $Q_2(G_P)$. In this subsection we want to define these graph transformation rules *correctly* in the sense that $Q_1(P)$ and $Q_2(G_P)$ are weak bisimilar (see Figure 6.11). The idea is to define the graph transformation rules for each IOS rule in a such way that a justification of each transition in Q_1 be equivalent to a process of application of the graph transformation rules.

The task to define the behavioural semantics of the CCS language with GTRs is not a simple one, because the IOS and the GTR techniques have different underlying principles. The main difference between the two semantics is that IOS rules are not always part of the transition system. The IOS rules such as Prefixing, Summation, CompositionI, CompositionII and Recursion serve only to justify a transition without being displayed by the transition system. The transition system generated by GTRs, on the contrary, allows to track every GTR being applied.

We illustrate the difference on a running example. For this, we show how inference rules of IOS affect a choice of a transition in LTS without being part of it. We generate an LTS for the process $P = a.b.nil + c.nil \mid d.nil$. The LTS is depicted in Figure 6.6. Here, the states are marked by intermediate processes, which will be simplified on the next step, and the transitions are labelled with the event that

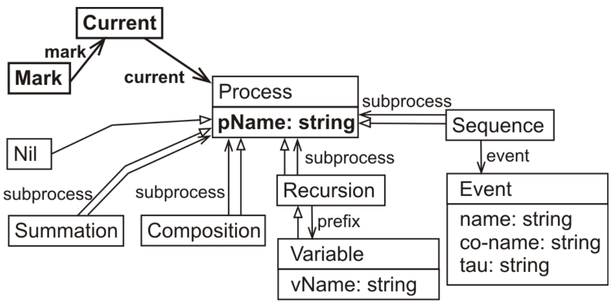


Figure 6.12: Run-time graph T^{rt}_{CSS} for the CCS model

occurs. The initial state is labelled with the original process P. The transition $P \xrightarrow{a} P'$ is possible if it is justified. The inference diagram for the transition is performed below.

$$\dfrac{\dfrac{}{a.b.nil \xrightarrow{a} b.nil}\ \mathsf{Prefixing}}{a.b.nil + c.nil \mid d.nil \xrightarrow{a} b.nil}\ \mathsf{Summation}$$

According to IOS, inference rule Summation must be applied to P, which reasons about a choice of subprocesses. The rule for Summation can be read as follows: if any one summand E_j of the sum $\sum_{i \in I}$ has an action, then the whole sum also has that action. Being not part of a transition system, but only reasoning about a transition, the Summation rule represents an invisible choice of the path in the transition system. Finally, we apply the Prefixing rule, which leads to the true statement. This means that transition $\xrightarrow{a}$ is possible from the state P.

To implement the IOS with graph transformations, we do the following. Rather than deletion of a syntactic structure, we use a pointer - a special node of type Current, which flows through a CCS graph, and on each step it points to a subtree, which has a corresponding process. The process of moving of a Current-node is similar to a justification process, when process P is simplified by inference rules on each step. Since we want to separate the cases for moving a Current-node from Summation-node, Composition-node and Recursion-node, we use an auxiliary pointer, a Mark-node. It helps to keep the LTSs Q_1 and Q_2 weak bisimilar, by separating the cases $P_1 + P_2$ and $P_1|P_2$, and avoiding a loop of invisible steps, e.g. for a process $a.nil + \mu x.x$.

In order to define the behavioural semantics by means of graph transformation rules we build a run-time graph by enhancing the type graph T^{st}_{CCS} (from Figure 6.8) with nodes of the Current and Mark types. For better understanding we also add a name to each Process-node, to identify a subprocess each node corresponds to. The run-time graph T^{rt}_{CCS} is presented in Figure 6.12.

Figure 6.13 illustrates the typed graph for the process $P = a.b.nil + c.nil \mid d.nil$ with dynamic elements. The Current-node is connected with the Process-node, which corresponds to the process $P = a.b.nil + c.nil \mid d.nil$.

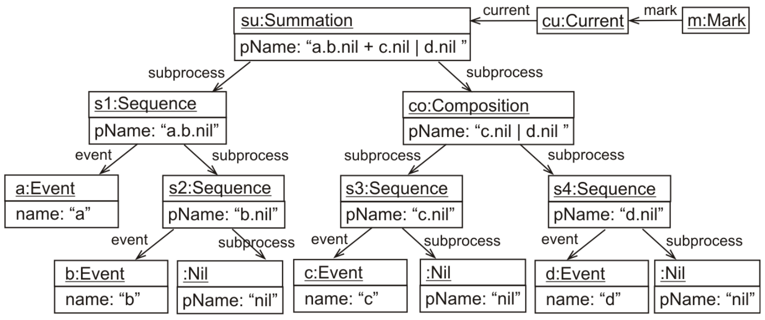


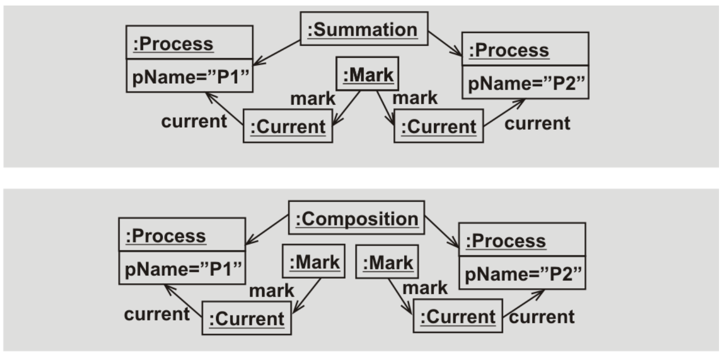
Figure 6.13: Typed graph for the CCS process $P = a.b.nil + c.nil|d.nil$

During the application of graph transformation rules, we want that the Current-node flows through the tree structured typed graph. The Current-node starts from the root node of a tree and flows in the direction to one of the leaves, possibly being duplicated. In case of recursion, the Current-node is moved from a leaf, which is a Variable-node, to a certain Recursion-node, which is connected with a Variable-node with the same attribute as the already mentioned Variable-node. If there exists an edge between a Current-node and a Process-node, denoted P_1 (Note: all nodes of Sequence, Composition, Summation, Recursion and Nil-types are also nodes of the Process-type), we say that the Current-node *points to* a Process-node or to a process P_1. The dynamic element Mark keeps the track of Current-node, thereby giving a special meaning to each Current-node.

We agree to use a special convention in order to distinguish the original CCS process and the CCS process defined as a graph. According to the EBNF, each process could be written as $P = P_1 + P_2$, $P = P_1 \mid P_2$, $\ldots$. Therefore, we can write $P_{P_1+P_2}$, $P_{P_1|P_2}$, $\ldots$, to denote an original process, and $G_{P_1+P_2}$, $G_{P_1|P_2}$, $\ldots$, to denote its corresponding graph, which emphasizes a syntactic structure. Let us consider a graph $G_{a.P_1} \subseteq G$, which is a corresponding graph for a process $a.P_1$. If there exists a single Current-node in a graph G, which points to a root node of $G_{a.P_1}$, then $G^{\mathsf{rt}}_{a.P_1} = G$. It is obvious that there exists a subgraph $G_{P1} \in G_{a.P_1}$, which is a corresponding graph for P_1. If the Current-node was moved to a root node of G_{P_1}, then we write $G = G^{\mathsf{rt}}_{P_1}$ and say that the rule turned graph $G^{\mathsf{rt}}_{a.P_1}$ into $G^{\mathsf{rt}}_{P_1}$ or transition $G^{\mathsf{rt}}_{a.P_1} \to G^{\mathsf{rt}}_{P_1}$ takes place.

It is evident, there is an additional state in the graph transformation system, in which the Current-node has been moved and duplicated, but the corresponding process has not been simplified. This happens after the Current-node flows through the Summation-nodes and Composition-nodes. We denote these intermediate states as $G^{\mathsf{rt}}_{P_1 \vee P_2}$ and $G^{\mathsf{rt}}_{P_1 \wedge P_2}$ respectively (see Figure 6.14). Operator $\wedge$ has higher priority than operator $\vee$.

Figure 6.14: Pattern for graphs, which represent additional states that do not have a corresponding process

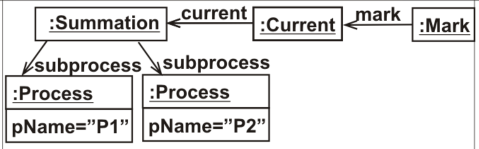
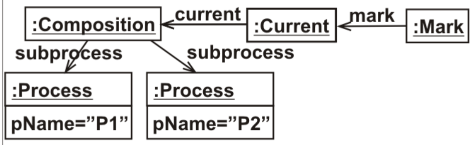
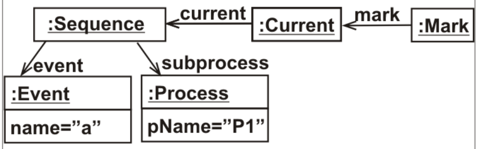
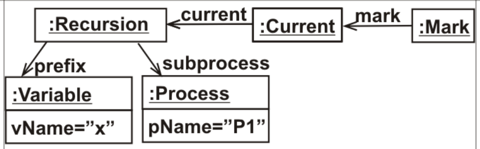
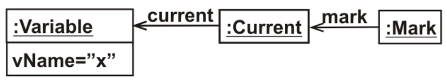
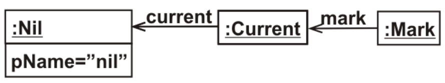


We consider the process $P = a.b.nil + c.nil\,|\,d.nil$ and a corresponding graph G_P. If we apply the graph transformation rules to this graph, we could get a state when there are three Current-nodes, which point at the subprocesses $P_1 = a.b.nil$, $P_2 = c.nil$ and $P_3 = d.nil$ and the corresponding graph is denoted as $G^{\mathsf{rt}}_{a.b.nil \vee c.nil \wedge d.nil}$. The label transition system generated by the graph transformation system for the graph $G_{a.b.nil+c.nil|d.nil}$ is shown below:

$$\begin{tikzcd}
 & G_{a.b.nil+c.nil|d.nil} \arrow[d, "ccsInitial"'] & & \\
 & G^{\mathsf{rt}}_{a.b.nil+c.nil|d.nil} \arrow[d, "ccsSummation"'] & & \\
 & G^{\mathsf{rt}}_{a.b.nil \vee c.nil|d.nil} \arrow[dl, "ccsSequence"'] \arrow[dr, "ccsComposition"] & & \\
G^{\mathsf{rt}}_{b.nil} \arrow[d, "ccsSequence"'] & & G^{\mathsf{rt}}_{a.b.nil \vee c.nil \wedge d.nil} \arrow[dl, "ccsSequence"'] \arrow[dr, "ccsSequence"] & \\
G^{\mathsf{rt}}_{nil} & G^{\mathsf{rt}}_{nil \wedge d.nil} \arrow[d, "ccsSequence"'] & & G^{\mathsf{rt}}_{c.nil \wedge nil} \arrow[d, "ccsSequence"'] \\
 & G^{\mathsf{rt}}_{nil \wedge nil} & & G^{\mathsf{rt}}_{nil \wedge nil}
\end{tikzcd}$$

The original IOS requires a justification of each transition, thereby simplifying the part of a process on each step of semantic execution. The GTRs do not change the syntactic part of the graph (elements typed over the T^{st}_{CSS}), instead the Current-node and Mark-node may change their position, fork or withdraw. The idea of simplification is that Current-node points at a Process-node which is the root of a subtree that corresponds to a certain subprocess. Therefore, we can define corresponding structures over the processes and graphs. In Table 6.1 there is an abbreviation for a process on each step of its simplification, its corresponding graph structure and the notation for a graph structure. Figure 6.15 illustrates the additional patterns for CCS graphs.

Table 6.1: CCS processes and their corresponding graph structures

CSS Process	Pattern for a Corresponding Graph	Corresponding Graph
P_{P1+P2}		G^{rt}_{P1+P2}
$P_{P1 \mid P2}$		$G^{rt}_{P1 \mid P2}$
$P_{a.P1}$		$G^{rt}_{a.P1}$
$P_{\mu x.P1}$		$G^{rt}_{\mu x.P1}$
P_x		G^{rt}_x
P_{nil}		G^{rt}_{nil}

We use $\mathcal{G}^{\mathsf{rt}}_{CCS}$ to denote the sets of all well-formed (run-time) CCS graphs typed over T^{rt}_{CCS}.

Finally, we present the graph transformation rules, which specify the behaviour of the CCS language by moving Current- and Mark-nodes. Each rule corresponds to

Figure 6.15: Additional pattern for a graph and corresponding notation

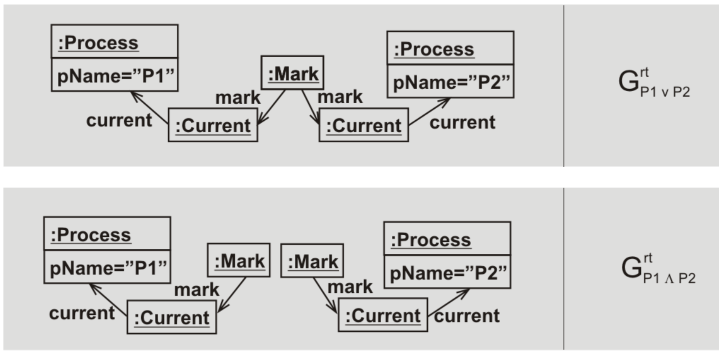


Table 6.2: The names of graph transformation rules for the CCS behavioural semantics and their corresponding rules from IOS semantics

Graph transformation rule name	**Corresponding IOS inference rule**
ccsInitial	None
ccsSequence	Prefixing $a.P \xrightarrow{a} P$
ccsCoAction	CompositionII $\frac{P\xrightarrow{a}P' \; Q\xrightarrow{\bar{a}}Q'}{P\|Q\xrightarrow{\tau}P'\|Q'}$
ccsSummation	Summation $\frac{P\xrightarrow{a}P'}{P+Q\xrightarrow{a}P'}$ $\frac{Q\xrightarrow{a}Q'}{P+Q\xrightarrow{a}Q'}$
ccsComposition	CompositionI $\frac{P\xrightarrow{a}P'}{P\|Q\xrightarrow{a}P'\|Q}$ $\frac{Q\xrightarrow{a}Q'}{P\|Q\xrightarrow{a}P\|Q'}$
ccsVariableDeclaration ccsRecursion	Recursion $\frac{P[\mu x P/x]\xrightarrow{a}P'}{\mu x P\xrightarrow{a}P'}$

one or two IOS rules (see Table 6.1.3), except the first one, which creates Current- and Mark-nodes for the syntax graph. The recursion is specified by two rules. The first moves a Current-node through the node, which corresponds to a prefix μ. The second rule accomplishes a substitution of a variable into a CCS process.

The rules are specified over the run-time graph T^{rt}_{CCS} and are presented in Figures 6.16-6.23. We define a semantic rule system as a partial mapping $\mathcal{RS}_{CCS}$: $\mathsf{Sym}_{CCS} \to \mathsf{Rule}_{CCS}$, where Rule_{CCS} is a set of graph transformation rules from Figures 6.16-6.23, $\mathsf{Sym}_{CCS} = \{$ccsInitial, ccsSequence, ccsCoAction, ccsSummation, ccsComposition, ccsVariableDeclaration, ccsRecursion$\}$ be the names in the rule system for the CCS graph. All of these rules affect the dynamic elements, but keep

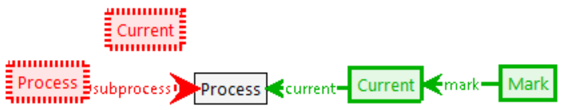


Figure 6.16: The ccsInitial rule (creates first dynamic elements in a static CCS graph)

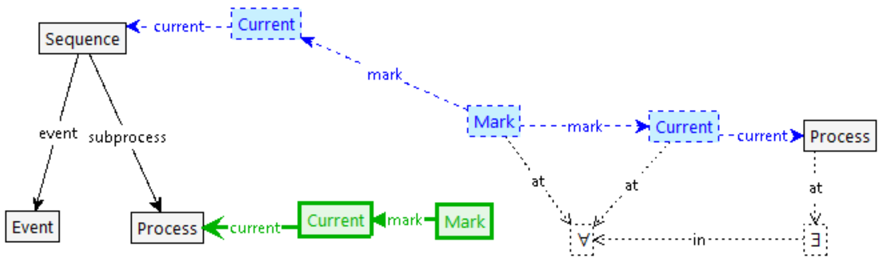


Figure 6.17: The ccsSequence rule template (moves a Current-node from a graph $G_{a.P}$ to a graph G_P)

the original structure. In the follow, each rule is separately explained.

The ccsInitial rule (see Figure 6.16) creates a Current-node and a Mark-node for a Process-node, which has no incoming *subprocess* edges, i.e. it is a root node.

Figure 6.17 illustrates a rule, which we call ccsSequence template. It is similar to IOS Prefixing rule, because it needs a match for a graph $G^{\mathsf{rt}}_{a.P}$ and it moves a Current-node from the graph $G_{a.P}$ to the graph G_P. However, the rule depicted in Figure 6.17 does not indicate which Event-node it is applied to. Since we want to separate the cases, when the rule is applied to different graphs, e.g. to graphs $G^{\mathsf{rt}}_{a.P}$ and $G^{\mathsf{rt}}_{b.P}$, here a and b are different events names, the rule must include an attribute value of the Event-node. For this, the template from Figure 6.17 must be extended with an attribute node, which has a specific value (specification of the event a). The name of the extended rule is derived from a prefix ccsSequence juxtaposed with an attribute value of the Event-node.

Figure 6.18 depicts the ccsSequenceBig rule, which is a result of the extension of the ccsSequence template. The Event-node is now with a specific attribute, which value is *big*. In the following we explain the ccsSequence template in detail.

The ccsSequence template requires the existence of a Sequence-node with two child nodes, one of them of a Process type, and a Current-node, which is connected with the Sequence-node. The rule creates a Current-node and a Mark-node such that there is an edge between the newly created Current-node and the Process-node and an edge between the newly created Current-node and the newly created Mark-node. In addition, the rule deletes the initial Current-node, the initial Mark-node and all Current-nodes that have a connection to the initial Mark-node. It allows to delete

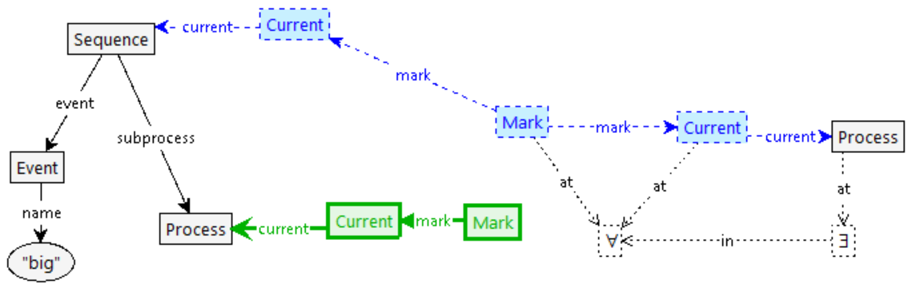


Figure 6.18: The ccsSequenceBig rule (moves a Current-node from a graph $G_{big.P}$ to a graph G_P) is an extension of the ccsSequence template with the attribute node for the Event-node with the value *big*

all Current-nodes created after application of the ccsSummation rule. Therefore, the ccsSequence rule makes a choice of a subprocess instead of the ccsSummation rule. It is expected that the ccsSequence rule performs the following transition:

$$G^{\mathsf{rt}}_{a.P} \xrightarrow{a} G^{\mathsf{rt}}_{P}$$

The ccsCoAction template (Figure 6.19) describes very similar transition step to a transition $a.P_1 \mid \bar{a}.P_2 \xrightarrow{\tau} P_1 \mid P_2$ from the original semantics, because it moves a Current-node from the graph $G^{\mathsf{rt}}_{a.P_1|\bar{a}.P_2}$ to the graph $G^{\mathsf{rt}}_{P_1|P_2}$. The template reminds the ccsSequence template, the difference is that the ccsCoAction template requires two Sequence-nodes instead of one. Additional condition is the following. One of the Sequence-nodes has a child Event-node with a *name*-attribute and a child Event-node with a *co-name*-attribute. The value of these attributes is the same. Moreover, for each Sequence-node there is a Current-node. There exists a Mark-node which connects these Current-nodes. It is expected that the ccsCoAction rule performs the following transition:

$$G^{\mathsf{rt}}_{a.P_1|\bar{a}.P_2} \xrightarrow{tau} G^{\mathsf{rt}}_{P_1 \vee P_2}$$

Note that the ccsCoAction template must be also extended (similarly to the ccsSequence template) with attributes values for the Event-node. The final rule must be added to a graph transformation system with an appropriate name (which is derived from a prefix ccsCoAction juxtaposed with an attribute value of the Event-node.).

Throughout the rest of this thesis, ccsSequence and ccsCoAction stand for any of the extended rules from the correspondent template.

The ccsSummation rule (Figure 6.20) is the most interesting among implemented rules. Rather than performing a choice (according to a description of the operator “+”) the rule duplicates Current-node for two child nodes of a root node of the graph

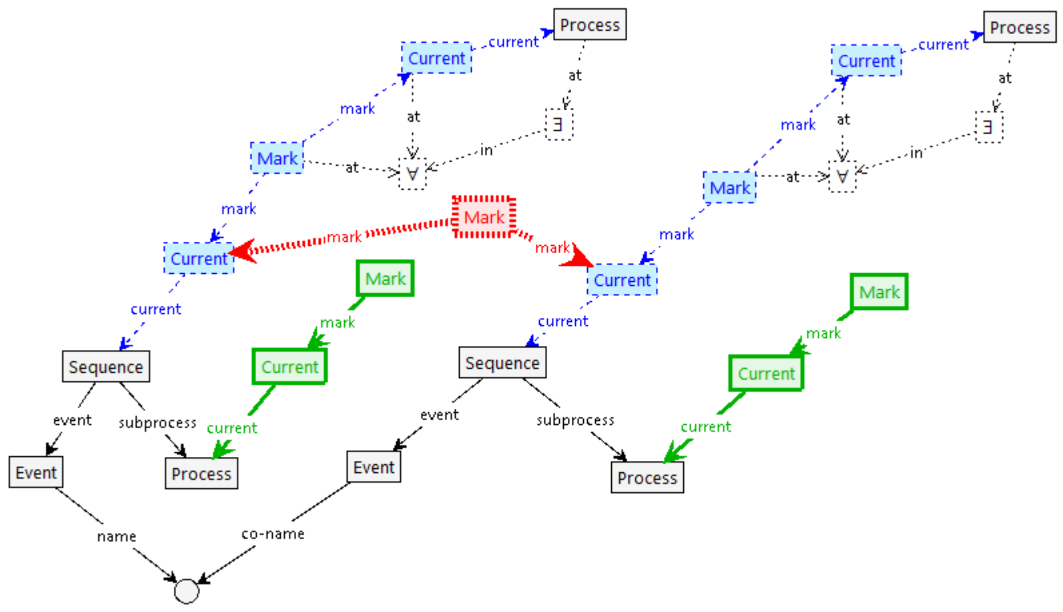


Figure 6.19: The ccsCoAction template (moves a Current-node from a graph $G^{\mathsf{rt}}_{a.P_1|\bar{a}.P_2}$ to a graph $G^{\mathsf{rt}}_{P_1|P_2}$)

$G^{\mathsf{rt}}_{P_1+P_2}$. Such an implementation allows to keep the weak bisimilar behaviour. So, if the rule matches the graph $G^{\mathsf{rt}}_{P_1+P_2}$, the ccsSummation rule does not move a Current-node to a particular child Process-node, instead it moves the Current-node to both Process-nodes. Therefore, the choice is deferred for the ccsSequence rule. It is expected that the ccsSummation rule performs the following transition:

$$G^{\mathsf{rt}}_{P_1+P_2} \xrightarrow{\tau} G^{\mathsf{rt}}_{P_1 \vee P_2}$$

The ccsComposition rule is similar to the original IOS CompositionI rule (Figure 6.21). The rule needs a match for a graph $G^{\mathsf{rt}}_{P_1|P_2}$ and then it duplicates Current-node and a Mark-node for two child nodes of a root node. In contrast to the ccsSummation rule, the CompositionI rule creates two Current-nodes with different Mark-nodes, which keep the connections of an original Mark-node. Then, if the ccsSequence rule is applied to one of the subprocesses, the second subprocess will be still executed. It is expected that the CompositionI rule performs the following transition:

$$G^{\mathsf{rt}}_{P_1|P_2} \xrightarrow{\tau} G^{\mathsf{rt}}_{P_1 \wedge P_2}$$

The ccsVariableDeclaration rule (Figure 6.22) is equivalent to the simplification of a process $\mu x.P$ to a process P. If we interpret this process for a graph, then the rule moves a Current-node from $G^{\mathsf{rt}}_{\mu x.P}$ to G^{rt}_{P}. However the ccsVariableDeclaration rule

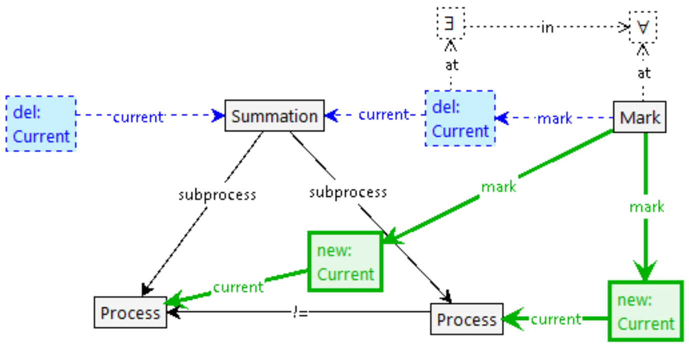


Figure 6.20: The ccsSummation rule (duplicates a Current-node for child Process-nodes)

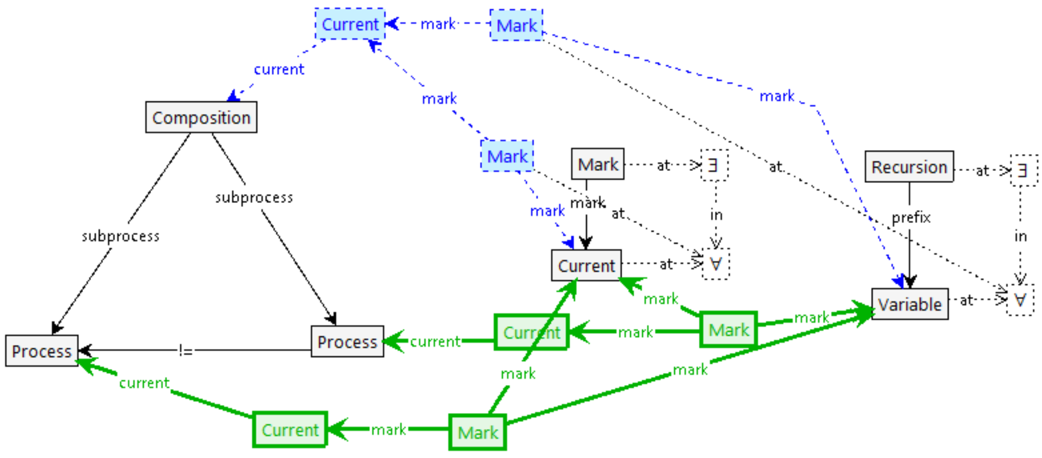


Figure 6.21: The ccsComposition rule (creates two Current-node with different Mark-nodes, the nearly created Mark-nodes keep the connections of an original Mark-node)

additionally keeps the connection to a node, where the variable x was firstly met or declared, i.e. there is a connection from a Recursion-node to a Variable-node. Being kept during the rule applications, this connection is used for recursion implementation. The connection is realized by a Mark-node and an edge to the Variable-node. It is expected that the ccsVariableDeclaration rule performs the following transition:

$$G^{\mathsf{rt}}_{\mu x.P} \xrightarrow{\tau} G^{\mathsf{rt}}_{P}$$

The ccsRecursion rule performs a recursive step. For the original IOS semantics, it means a substitution of a variable x with a process P, i.e. $P[\mu x P/x]$. If a Current-node reaches a Variable-node, the rule moves the Current-node to the place, where the variable was declared (by the ccsVariableDeclaration rule). It is realized due to

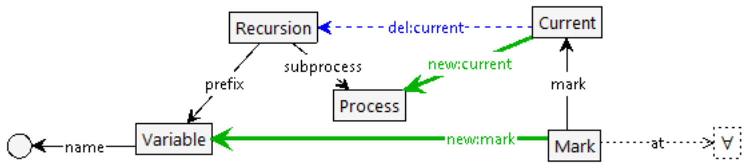


Figure 6.22: The ccsVariableDeclaration rule (moves a Current-node from $G^{\mathsf{rt}}_{\mu x.P}$ to G^{rt}_{P} and additionally keeps the connection to a node, where the variable x was declared)

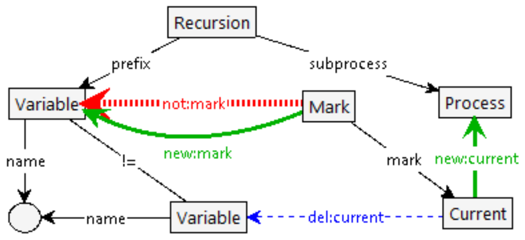


Figure 6.23: The ccsRecursion rule (performs a recursive step)

a connection of a Mark-node with a Variable-node at the moment of its declaration. The connection is kept during the semantics execution. It is expected that the ccsRecursion rule performs the following transition:

$$G^{\mathsf{rt}}_{x} \xrightarrow{\tau} G^{\mathsf{rt}}_{P}$$

6.1.4 Semantics Preservation

To be sure that the rule system $\mathcal{RS}_{CCS}$ describes the same behaviour as the original semantics, we use the equivalence over an LTS generated by the original IOS and an LTS generated by the graph transformation system. In this subsection we want to show that an LTS generated for any process P is weak bisimilar (see Chapter 4) to an LTS generated for the graph G_P.

Let Q_1 be an LTS such that $Q_1 = \langle S, \rightarrow, i, Act\rangle$ with a set of labels $Act = \{a, b, c, \ldots\}$, where Act is some set of names. An LTS generated for the process P by IOS is denoted as $Q_1(P)$. Let Q_2 be an LTS such that $Q_1 = \langle S, \rightarrow, i, L\rangle$ with a set of labels $L = \{$ccsInitial, ccsSequence, ccsCoAction, ccsSummation, ccsComposition, ccsVariableDeclaration, ccsRecursion$\}$, where elements of the set L are the names of the graph transformation rules from $\mathcal{RS}_{CCS}$. Note that here, ccsSequence and ccsCoAction stand for sets, which consist of all possible extensions from the respective templates for each label from Act. An LTS generated for the graph G_P by the graph transformation rules is denoted as $Q_2(G_P)$. Our task is to compare $Q_1(P)$ and $Q_2(G_P)$ and to show that they are weak bisimilar.

There is an obvious problem: the LTSs do not have the same labels. Moreover, an LTS generated by IOS is labelled with event names. To map the labelling sets we do the following. We use a common name for every transition in an LTS and write Event for each transition $\xrightarrow{a}$ in order to establish correspondence between LTSs transitions. We also map the labels from an LTS generated by graph transformation rules to be compared to a common set of names. Since each graph transformation rule has a corresponding inference rule, we provide a mapping based on an inference diagram, where the inference rules are applied in a random order, however $\mathsf{Prefixing}$ rule and $\mathsf{CompositionII}$ rule for concurrent step are always at the end. This observation allows us to map each rule to an invisible step except the last ones of an inference process:

$$
\begin{array}{lll}
\mathit{map}_1\colon & \forall a \in \mathit{Act}\backslash\tau & a \mapsto \mathsf{EventA}, \\
 & \tau \in \mathit{Act} & \tau \mapsto \mathsf{EventTau}, \\
\mathit{map}_2\colon & \forall a \in \mathit{Act}\backslash\tau & \mathsf{ccsSequenceA} \mapsto \mathsf{EventA} \\
 & \forall a \in \mathit{Act}\backslash\tau & \mathsf{ccsCoActionA} \mapsto \mathsf{EventTau}, \\
 & \mathsf{ccsSummation} \mapsto \tau & \\
 & \mathsf{ccsComposition} \mapsto \tau, & \\
 & \mathsf{ccsRecursion} \mapsto \tau, & \\
 & \mathsf{ccsVariableDeclaration} \mapsto \tau, & \\
 & \mathsf{ccsInitial} \mapsto \tau. &
\end{array}
$$

Here, $\mathit{Act}\backslash\tau$ is a set Act without τ.

Note that EventA is a different label for every $a \in \mathit{Act}$ (e.g. for $b \in \mathit{Act}$ $b \mapsto \mathsf{EventB}$). Therefore, it is implied that for every label $a \in \mathit{Act}$ there exist a separate $\mathsf{ccsSequenceA}$ rule and a separate $\mathsf{ccsCoActionA}$ rule. Therefore, the function $\mapsto$ maps not all events to the same label Event, but each label to a different rule from the CCS semantic system. It means that the event $a \mapsto \mathsf{EventA}$ and $\mathsf{ccsSequenceA} \mapsto \mathsf{EventA}$, $\tau \mapsto \mathsf{EventTau}$ (here, $\tau \in \mathit{Act}$) and $\mathsf{ccsCoActionA} \mapsto \mathsf{EventTau}$.

Let be $\mathsf{Sym} = \{\mathsf{Event}, \tau\}$ a common set of names for the rules from $\mathit{dom}(\mathcal{RS}_{CCS})$ and Act, here τ stays for the internal step. We call such mappings $\mathit{map}_1 : \mathit{Act} \rightarrow$

Sym and $map_1 : dom(\mathcal{RS}_{CCS}) \to \mathsf{Sym}$ *non-trivial*, if it does not map every rule name to τ.

We also define the mappings over LTSs, which affect only labelling sets according to the mapping defined above. So, $map : \mathcal{P}(Q) \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is a universe of transition systems, if $Q = \langle S, \to, i, L\rangle$, then $map(Q) = \langle S, \to, i, L'\rangle$, where $l' = map(l)$, $l \in L$, $l' \in L'$.

To show the behavioural preservation, i.e. the preservation of behavioural properties, we need to show the weak bisimulation for LTSs Q_1 and Q_2 with respect to the defined mappings: $map_1(Q_1(P)) \approx map_2(Q_2(G_P))$. This will imply the behavioural preservation (see Chapter 4).

Further, we define an equivalence relation $\mathcal{R}_{CCS}$ (i.e. $\approx$) over the states of the LTSs $Q_1(P)$ and $Q_2(Q_P)$ and prove that the defined relation is a weak bisimulation. At first, we introduce some notation and observations about CCS processes and corresponding graphs, which will help us to define the relation $\mathcal{R}_{CCS}$.

Auxiliary notation

We start with notation. For Label-node (a node of a Label type) we write v_{Label}. We shorten the names of some types to the first three-four letters, like for example Summation we write shortly v_{Sum}. For an edge e labelled label going from a node v to v', we simply write $\mathsf{label}(v, v')$. We also use these as predicates.

We define the notation for the nodes that could be connected with Current-node and Mark-node during the semantic execution:

$current(v_{Proc})$, if there exists a Current-node v_{Curr} and a Process-node v_{Proc} such that $current(v_{Curr}, v_{Proc})$.

$mark(v_{Curr})$, if there exists a Mark-node v_{Mark} and a Current-node v_{Curr} such that $current(v_{Mark}, v_{Curr})$.

Additionally, we provide the notation for Process-nodes. Since a CCS graph has a tree structure, then we define the notation for a root element and leaves. For a Process-node v, we write:

$root(v)$, if $\neg\exists$ Process-node v_0 s.t. $subprocess(v_0, v)$.
$leaf(v)$, if $\neg\exists$ Process-node v' s.t. $subprocess(v, v')$.

Since there are the cases, when there could be more than one Current-node in a run-time graph, we need precisely to define its index. We formalize the notion for the index of CCS graphs typed over T^{rt}_{CCS}.

Proposition 6.1.1. *Let G_{P_1} and $G_{P_2} \in \mathcal{G}^{\mathsf{st}}_{CCS}$ be well-formed CCS graphs and $G^{\mathsf{rt}} \in \mathcal{G}^{\mathsf{rt}}_{CCS}$ be a run-time typed over T^{rt}_{CCS} graph, such as G_{P_1}, $G_{P_2} \subseteq G^{\mathsf{rt}}$. Let also v^1_{Proc} and v^2_{Proc} be root nodes in the tree-structured graphs G_{P_1} and G_{P_2}, respectively. For every couple of* Current*-nodes $v^1_{Curr}, v^2_{Curr} \in G^{\mathsf{rt}}$ such that $current(v^1_{Curr}, v^1_{Proc})$*

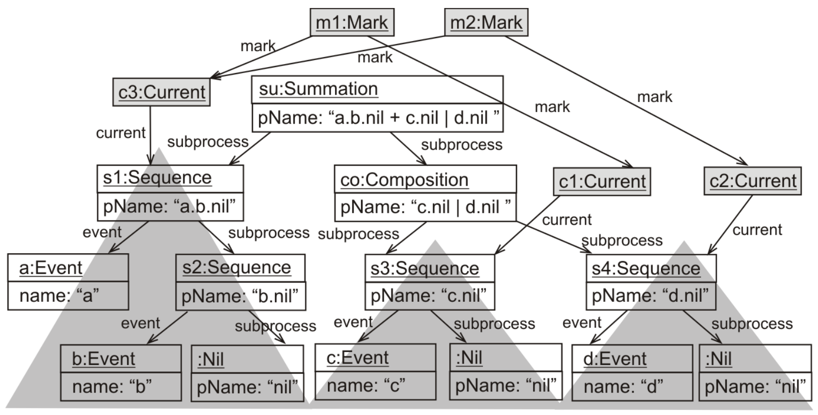


Figure 6.24: A corresponding run-time graph for the process $P = a.b.nil + c.nil \mid d.nil$, which demonstrates the definition of a corresponding graph index

and $current(v^2_{Curr}, v^2_{Proc})$ *one of the following conditions hold, which is a result of the graph index:*

- *if there exists a* Mark*-node* v_{Mark} *such that* $mark(v_{Mark}, v^1_{Curr})$ *and* $mark(v_{Mark}, v^2_{Curr})$*, then we write* $G^{\mathsf{rt}}_{P_1 \vee P_2} \subseteq G^{\mathsf{rt}}$*.*
- *if there exist two* Mark*-nodes* v^1_{Mark} *and* v^2_{Mark} *such that* $mark(v^1_{Mark}, v^1_{Curr})$ *and* $mark(v^2_{Mark}, v^2_{Curr})$*, then we write* $G^{\mathsf{rt}}_{P_1 \wedge P_2} \subseteq G^{\mathsf{rt}}$*.*

The notation for three and more Current-nodes is defined by the induction, on the assumption that the conditions described above hold for any couple of Current-nodes.

We provide again an example for the graph $G^{\mathsf{rt}}_{a.b.nil \vee c.nil \wedge d.nil}$ in Figure 6.24 which corresponds to process $P = a.b.nil + c.nil \mid d.nil$. There are three grey triangles which emphasize subgraphs $G_{a.b.nil}$, $G_{c.nil}$, $G_{d.nil}$. The root nodes of these graphs are connected with Current-nodes which are by turn connected to Mark-nodes. The index of the graph $G^{\mathsf{rt}}_{a.b.nil \vee c.nil \wedge d.nil}$ was received after analyzing the graph and the conditions described above.

Observations

To provide later a proof we need some observations about the syntactic structure of CCS graphs. We continue with observations concerning the correspondences between CCS process and CCS graphs. The proof results from the definition of well-formed CCS graphs. A number of further results show that for each CCS process there exists a corresponding CCS graph and vice versa.

Proposition 6.1.2. *Let P be a CCS process then there exists a corresponding graph $G_P \in \mathcal{R}^{\mathsf{st}}_{CCS}$, s.t if there exists another corresponding graph $G' \in \mathcal{R}^{\mathsf{st}}_{CCS}$ then $Q(G) \approx Q(G')$.*

(Seq) Let $P = a.P_1$ and $G_{a.P_1}$ be a corresponding graph such that a Sequence-*node is a root node with an attribute value "a" in the tree structured graph $G_{a.P_1}$. Then the* Sequence-*node has a child* Process-*node, which is a root of a tree structured subgraph G_{P_1} (a corresponding graph for the process P_1).*

(Sum) Let $P = P_1 + P_2$ and $G_{P_1+P_2}$ be a corresponding graph such that a Summation-*node is a root node in the tree structured graph $G_{P_1+P_2}$. Then the* Summation-*node has two child* Process-*nodes which are roots of tree structured subgraphs G_{P_1} and G_{P_2} (corresponding graphs for the processed P_1 and P_2, respectively).*

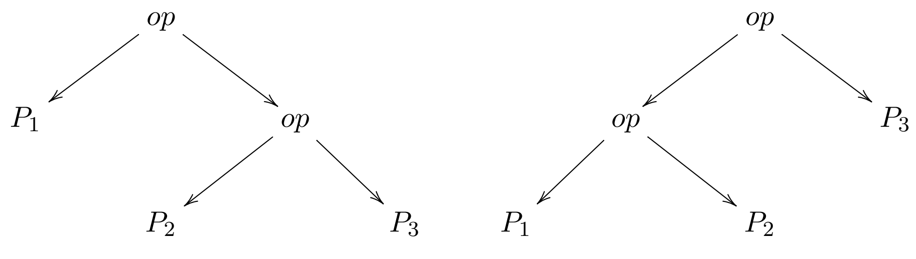
(Com) Let $P = P_1 \mid P_2$ and $G_{P_1|P_2}$ be a corresponding graph such that a Composition-*node is a root node in the tree structured graph $G_{P_1|P_2}$. Then the* Composition-*node has two child* Process-*nodes which are roots of tree structured subgraphs G_{P_1} and G_{P_2} (corresponding graphs for the processed P_1 and P_2, respectively).*

(Nil) Let $P = nil$ and G_{nil} be a corresponding graph such that a Nil-*node is a root (and also a leaf) node in the tree structured graph G_{nil}. Then the* Nil-*node has no child nodes.*

(Rec) Let $P = \mu x.P_1$ and $G_{\mu x.P_1}$ be a corresponding graph such that a Recursion-*node is a root node in the tree structured graph $G_{\mu x.P_1}$. Then the* Recursion-*node has a child* Process-*node which is a root of a tree structured subgraph G_{P_1} (a corresponding graph for the processed P_1).*

(Var) Let $P = x$ and G_x be a corresponding graph such that a Variable-*node is a root (and also a leaf) node in the tree structured graph G_x. Then the* Variable-*node has no child nodes.*

Proof. Although a corresponding graph for a process P could have different syntactic representations (e.g. there are exists two corresponding graphs $G_{(P_1+P_2)+P_3}$ and $G_{P_1+(P_2+P_3)}$ for a process $P = P_1 + P_2 + P_3$), the behavioural model specified as an LTS has the same meaning. It means that the LTSs generated for different corresponding graphs of process P are weak bisimilar. Let consider graphs $G_{P_1+P_2+P_3}$ and $G_{P_1|P_2|P_3}$, they could have two different syntactic structures, as it is shown below, where a tree structure of two possible graphs is represented.



$$\begin{tikzcd}
 & op \arrow[dl] \arrow[dr] & & \\
P_1 & & op \arrow[dl] \arrow[dr] & \\
 & P_2 & & P_3
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & op \arrow[dl] \arrow[dr] & \\
 & op \arrow[dl] \arrow[dr] & & P_3 \\
P_1 & & P_2 &
\end{tikzcd}$$

$Case1$ $Case2$

here, the operators $+$ and $|$ are denoted as *op*.

If we apply $\mathcal{RS}_{CCS}$ to both syntactic structures, the LTSs generated for both graphs are weak bisimilar. The implementation of the ccsSummation and ccsComposition rules makes it possible to perform an invisible steps in an LTS, which do not affect the weak bisimulation. As you can see on the example of corresponding graph for the process $P = P_1 + P_2 + P_3$, there are two LTSs:

$$\begin{tikzcd}
& G^{\mathsf{rt}}_{P_1+(P_2+P_3)} \arrow[d, "\tau"'] & & & & G^{\mathsf{rt}}_{(P_1+P_2)+P_3} \arrow[d, "\tau"'] & \\
& G^{\mathsf{rt}}_{P_1\vee(P_2+P_3)} \arrow[dl, "P_1"'] \arrow[d, "\tau"'] & & & & G^{\mathsf{rt}}_{(P_1+P_2)\vee P_3} \arrow[d, "\tau"'] \arrow[dr, "P_3"] & \\
G^{\mathsf{rt}}_{P_1'} & G^{\mathsf{rt}}_{P_1\vee(P_2\vee P_3)} \arrow[dl, "P_1"'] \arrow[d, "P_2"'] \arrow[dr, "P_3"] & & \approx & & G^{\mathsf{rt}}_{(P_1\vee P_2)\vee P_3} \arrow[dl, "P_1"'] \arrow[d, "P_2"'] \arrow[dr, "P_3"] & G_{P_3'} \\
G^{\mathsf{rt}}_{P_1'} & G^{\mathsf{rt}}_{P_2'} & G^{\mathsf{rt}}_{P_3'} & & G^{\mathsf{rt}}_{P_1'} & G^{\mathsf{rt}}_{P_2'} & G^{\mathsf{rt}}_{P_3'}
\end{tikzcd}$$

here, a transition labelled as τ correspond to the applied ccsSummation rule. The transition labels P_1, P_2, P_3 mean that the transition is dependent from the inner structure of the corresponding grapps. The following states are weak bisimilar:

$$G^{\mathsf{rt}}_{P_1+(P_2+P_3)} \approx G^{\mathsf{rt}}_{(P_1+P_2)+P_3} \approx G^{\mathsf{rt}}_{P_1\vee(P_2+P_3)} \approx G^{\mathsf{rt}}_{(P_1+P_2)\vee P_3} \approx G^{\mathsf{rt}}_{P_1\vee(P_2\vee P_3)} \approx G^{\mathsf{rt}}_{(P_1\vee P_2)\vee P_3}$$

The proof of the adjustment for the graph structures follows by analysis of a well-formed patterns definition. □

Proposition 6.1.3. *Let $G \in \mathcal{G}^{\mathsf{st}}_{CCS}$ be a well-formed CCS graph, then there exists a CCS process P, such that the graph $G = G_P$ is a corresponding graph for the process P.*

(Seq) Let v_{Seq} be a Sequence*-node with an attribute value "a" and v_{Proc} be a* Process*-node in $G \in \mathcal{G}^{\mathsf{rt}}_{CCS}$, s.t. $root(v_{Seq})$ and $subprocess(v_{Seq}, v_{Proc})$, then there exists a process $P = a.P_1$, s.t $G = G_{a.P_1}$ and v_{Proc} is a root of the tree structured subgraph G_{P_1} (a corresponding graph for the process P_1).*

(Sum) Let v_{Sum} be a Summation*-node and v^1_{Proc}, v^2_{Proc} be* Process*-nodes in G, s.t. $root(v_{Sum})$ and $subprocess(v_{Sum}, v^1_{Proc})\wedge$ $subprocess(v_{Sum}, v^2_{Proc})$, then there exists a process $P = P_1 + P_2$, s.t. $G = G_{P_1+P_2}$ and v^1_{Proc},v^2_{Proc} are roots of the tree structured subgraphs G_{P_1} and G_{P_2} (corresponding graphs for the processed P_1 and P_2 respectively).*

(Com) Let v_{Com} be a Composition*-node and v^1_{Proc}, v^2_{Proc} be* Process*-nodes in G, s.t. $root(v_{Com})$ and $subprocess(v_{Com}, v^1_{Proc}) \wedge$ $subprocess(v_{Com}, v^2_{Proc})$, then there exists a process $P = P_1 \mid P_2$, s.t. $G = G_{P_1|P_2}$ and v^1_{Proc}, v^2_{Proc} are roots of the tree structured subgraphs G_{P_1} and G_{P_2} (corresponding graphs for the processed P_1 and P_2 respectively).*

(Nil) Let v_{Nil} be a Nil*-node in G, s.t. $root(v_{Nil})$, then there exists a process $P = nil$, s.t. $G = G_{nil}$.*

(Rec) Let v_{Rec} be a Recursion*-node and v_{Proc} be a* Process*-node in G, s.t. $root(v_{Rec})$ and $subprocess(v_{Rec}, v_{Proc})$, then there exists a process $P = \mu x.P_1$, s.t. $G = G_{\mu x.P_1}$ is a corresponding graph for a process P and G_{P_1} is a corresponding graph for the processes P_1 and $P_1[\mu x P_1/x]$.*

(Var) Let v_{Var} be a Variable*-node in G, s.t. $root(v_{Var})$, then there exists a process $P = P[\mu x P/x]$ with a corresponding graph G_P, where $root(v_{Proc})$, $G \subseteq G_P$ and there exists a* Recursion*-node v_{Rec}, which is connected with a* Variable*-node (i.e. $variable(v_{Rec}, v_{Var})$) that has the same attribute value as v_{Var}.*

Proof. Every process P is defined by the EBNF grammar, which was fully transformed into a well-formed CCS graph structure. Therefore, for each process P there exists a well-formed CCS graph. The proof for the adjusments follows by analysis of a well-formed patterns definition. □

Equivalence relation

After we defined the auxiliary notation and provided some observations, we have all necessary tools to define an equivalence relation $\mathcal{R}_{CCS}$ over the states of the LTSs Q_1 and Q_2. Recall, that the states of the LTS Q_1 consist of processes P defined by the EBNF grammar and transitions $\rightarrow$ are defined by the rules of inference (see Subsection 6.1.1 for more details). The states of the LTS Q_2 are graphs $G^{\mathsf{rt}} \in \mathcal{G}^{\mathsf{rt}}_{CCS}$ and transitions are defined with the rule system $\mathcal{RS}_{CCS}$. Therefore, we define a relation $\mathcal{R}_{CCS}$ over processes P and well-formed CCS graphs.

We use the previously defined notation for the processes and corresponding graphs. Additionally, we use a notion of different levels of subprocesses in respect to operators. This notation allows us to specify $\mathcal{RS}_{CCS}$ inductively. We say that a subprocess P' is from the *first level* of a process P, if P' is connected with other subprocesses of P only by $+$ and $|$. For example, the processes $a.b.nil$, $c.nil$, $d.nil$ are the only subprocesses from the first level of the process $P = a.b.nil + c.nil \,|\, d.nil$. We say that subprocess P'' is from the *second level* of process P, if P'' is connected with other subprocesses of P only by $\mu V.$ and maybe by $+$ and $|$. For example, the processes $\mu y.a.y + x$, $a.y$ and x are the only subprocesses from the second level of the process $P = \mu x.(\mu y.a.y + x)$.

We say that process P and graph G are in the relation $\mathcal{R}_{CCS}$, if (0) $G = G_P \in \mathcal{G}^{\mathsf{st}}_{CCS}$ is a corresponding graph for a process P, or for all subprocesses from the first level of P the following conditions hold:

1. $a.P_1 \subseteq P \Leftrightarrow \exists G_{a.P_1}$ s.t. $G_{a.P_1}$ is a corresponding graph for $P_{a.P_1}$, $G_{a.P_1} \subseteq G$ and $G^{\mathsf{rt}}_{a.P_1} \subseteq G$,
2. $P_1 + P_2 \subseteq P \Leftrightarrow$ One of the following conditions hold:
   [2a] $\exists G_{P_1+P_2}$ s.t. $G_{P_1+P_2}$ is a corresponding graph for $P_{P_1+P_2}$, $G_{P_1+P_2} \subseteq G$ and $G^{\mathsf{rt}}_{P_1+P_2} \subseteq G$,
   [2b] $\exists G_{P_1+P_2}$ s.t. $G_{P_1+P_2}$ is a corresponding graph for $P_{P_1+P_2}$, $G_{P_1+P_2} \subseteq G$ and $G^{\mathsf{rt}}_{P_1 \vee P_2} \subseteq G$,
3. $P_1 \mid P_2 \subseteq P \Leftrightarrow$ One of the following conditions hold:
   [3a] $\exists G_{P_1|P_2}$ s.t. $G_{P_1|P_2}$ is a corresponding graph for $P_{P_1|P_2}$, $G_{P_1|P_2} \subseteq G$ and $G^{\mathsf{rt}}_{P_1|P_2} \subseteq G$,
   [3b] $\exists G_{P_1|P_2}$ s.t. $G_{P_1|P_2}$ is a corresponding graph for $P_{P_1|P_2}$, $G_{P_1|P_2} \subseteq G$ and $G^{\mathsf{rt}}_{P_1 \wedge P_2} \subseteq G$,
4. $nil \subseteq P \Leftrightarrow \exists G_{nil}$ s.t. G_{nil} is a corresponding graph for P_{nil}, $G_{nil} \subseteq G$ and $G^{\mathsf{rt}}_{nil} \subseteq G$,
5. One of the following conditions hold:
   [5a] $\mu x.P_1 \subseteq P \Leftrightarrow \exists G_{\mu x.P_1}$ s.t. $G_{\mu x.P_1}$ is a corresponding graph for $P_{\mu x.P_1}$, $G_{\mu x.P_1} \subseteq G$ and $G^{\mathsf{rt}}_{\mu x.P_1} \subseteq G$,
   [5b] $P_1 \subseteq P$, where P_1 is a subprocess of the second level $\Leftrightarrow \exists G_{P_1}$ s.t. G_{P_1} is a corresponding graph for P_{P_1}, $G_{P_1} \subseteq G$ and $G^{\mathsf{rt}}_{P_1} \subseteq G$,
   [5c] $P_1 \subseteq P$, where P_1 is a subprocess of the second level, then for all subprocesses for the first level of P_1 conditions (1)-(6) hold $\Leftrightarrow \exists G_{P_1}$ - a corresponding graph for P_1, $G^{\mathsf{rt}}_{P_1}$ and all the conditions (1)-(6) hold for $G^{\mathsf{rt}}_{P_1}$.
6. $x \subseteq P \Leftrightarrow \exists G_x$ s.t. G_x is a corresponding graph for P_x, $G_x \subseteq G$ and $G^{\mathsf{rt}}_x \subseteq G$.

For example, if we have a process $P = P_1 + P_2 \mid P_3$, then the graph $G^{\mathsf{rt}}_{P_1 \vee P_2 \wedge P_3}$ is in relation with P, the same as the graphs $G^{\mathsf{rt}}_{P_1 \vee P_2|P_3}$ and $G^{\mathsf{rt}}_{P_1+P_2|P_3}$.

The relation $\mathcal{R}_{CCS}$ contains all pairs of CCS processes which are produced from the EBNF form and corresponding graphs, which structure was defined in Table 6.1 and in Figure 6.14.

Theorem 6.1.4. Given relation $\mathcal{R}_{CCS}$ (defined in above). Let P be a CCS process and G_P be a corresponding CCS graph. Then relation $\mathcal{R}_{CCS}$ is a weak bisimulation ($\approx$), i.e.

$$map_1(Q_1(P)) \approx map_2(Q_2(G_P))$$

Proof. of the Theorem 6.1.4 We need to show the property of mutual simulation for relation $\mathcal{R}_{CCS}$. For this, we provide a proof by induction, where the basis of induction is the requirement of initial states being in the relation. The initial states of the LTSs are the process P_0 and a corresponding graph $G_{P_0} \in \mathcal{G}^{\mathsf{st}}_{CCS}$. They satisfy the condition (1) for $\mathcal{R}_{CCS}$.

The assumption in the inductive step is that the statement holds for some subprocess P of a process P_0 and a subgraph G_P of a graph G_{P_0}, i.e. we assume $(P, G) \in \mathcal{R}_{CCS}$. To perform an inductive step, we prove the statement for the transition $P \xrightarrow{r} P'$, which is justified by the inference rules (see Section 6.1.1). We need to show that there is some G' such that $G \overset{\widehat{r}}{\Longrightarrow} G'$ with $(P', G') \in \mathcal{R}_{CCS}$.

The proof is based on the analysis of the syntactic structure of P. According to the EBNF grammar for CCS processes and the inference rules, the syntactic structure of process P is either $P_1 + P_2$, $P_1|P_2$, $\mu x.P$, $P[\mu x.P/x]$, $a.P_1$, or $a.P_1|\bar{a}.P_2$. Then the inference process can be described as follows: to the initial process P we apply the rules of inference, which step-by-step simplify it by replacing it with one of the subprocesses until one of the subprocesses is either $a.P$ or $a.P_1 \,|\, \bar{a}.P_2$ (otherwise the transition is not possible). We show that the inference process for the process P could be matched to the process of the GTR application to a graph G_P, as the result we always get a weak bisimilar transition step. For this, we consider P as initial process. which is replaced on the next steps by its subprocesses. We write $P \succ P_1 + P_2$ that means that the process P has a subprocess $P_1 + P_2$ wot which one of the inference rules will be applied as a next step.

Remark 1. In respect to the definition of $\mathcal{R}_{CCS}$ the graph G is either G_P or G_P^{rt}. In the first case we can obviously apply only the ccsInitial rule (see Figure 6.16), which performs an invisible step in a transition system, and we have graph G_P^{rt}, which is in relation $\mathcal{R}_{CCS}$ with P. Therefore, we consider later that $G = G_P^{\mathsf{rt}}$.

Remark 2. If a graph $G_P \in G$ is a corresponding graph for a process P and there exists the only Current-node, which points to a root node of the graph G_P, i.e. $G = G_P^{\mathsf{rt}}$, then $(P, G_P^{\mathsf{rt}}) \in \mathcal{R}_{CCS}$. According to the definition of the CCS process by the EBNF grammar, the structure of the process P is predefined. The proof follows by induction on the inner structure of P. For example, if $P = a.P_1$ then a corresponding graph is $G_{a.P_1}$. If there is a Current-node, then we have a graph $G_{a.P_1}^{\mathsf{rt}}$. The condition (1) holds for G and P. Since there are no other Current-nodes, the rest conditions of $\mathcal{R}_{CCS}$ hold for the process P and the graph G.

A short summary of the idea for the inductive proof is presented below. Arrows with solid lines represent transitions in LTSs $Q(P)$ (in the left) and $Q(G_P)$ (in the right), dotted line connects a process and a graph, which are in relation $\mathcal{R}_{CCS}$ (denoted as $\mathcal{R}$ in the Figure). There are some explanations in the right concerning a dotted line.

$$\begin{tikzcd}
P_0 \arrow[r, dotted, no head, "\mathcal{R}"] \arrow[d] \arrow[dr, dotted, no head, "\mathcal{R}"] & G^{\mathsf{st}}_{P_0} \arrow[d, "\tau"] & \text{- basis for induction} \\
\cdots \arrow[d] & G^{\mathsf{rt}}_{P_0} \arrow[d] & \text{- Remark 1} \\
\cdots \arrow[d] & \cdots \arrow[d] & \\
P \arrow[r, dotted, no head, "\mathcal{R}"] \arrow[d, "r"'] & G^{\mathsf{rt}}_{P} \arrow[d, Rightarrow, "r"] & \text{- inductive assumption} \\
P' \arrow[r, dotted, no head, "\mathcal{R}"] & G^{\mathsf{rt}}_{P'} & \text{- we show the existence of } G^{\mathsf{rt}}_{P'}
\end{tikzcd}$$

As we are looking at the LTSs with labels renamed according map_1 and map_2, in principle $r \in Act$ and $map_1(r) = \mathsf{Event}$. However, as we are interested in the particular semantic rule applied during the step, we will look at the original LTSs and show that map_1 and map_2 map rule names to the same label.
The proof proceeds depending on the syntactic structure of process P:

Case $P \succ P_1 + P_2$. Due to our assumption that $(P, G) \in \mathcal{R}_{CCS}$, G includes the corresponding graph for the process $P_1 + P_2$. Then according to (2), G includes either $G^{\mathsf{rt}}_{P_1 \vee P_2}$ or $G^{\mathsf{rt}}_{P_1 + P_2}$. In the first case the proof proceeds by the induction on the inner structure of P_1 or P_2. In the second case we proceed with arguing that the ccsSummation rule can be applied and that the resulted graph is still in the relation.

By Proposition 6.1.2 (Sum) there exists a graph $G_{P_1+P_2}$, which is a corresponding graph for P and there are subgraphs $G_{P_1}, G_{P_2} \subseteq G_{P_1+P_2}$, which are corresponding graphs for the subprocesses P_1 and P_2, respectively. By (2a) and Remark 1 there exists a Current-node in G, which points to the subgraph $G_{P_1+P_2}$.

We can construct a match for the ccsSummation rule in $G^{\mathsf{rt}}_{P_1+P_2}$ graph (see Figure 6.20 and Table 6.1). In the resulting graph G', the Current-node v_{Curr} is deleted and two Current-nodes are created: v^1_{Curr} and v^2_{Curr} such that $current(v^1_{Curr}, v^1_{Proc})$ and $current(v^2_{Curr}, v^2_{Proc})$. The resulting graph G' includes $G^{\mathsf{rt}}_{P_1 \vee P_2}$ (see Figure 6.14). The condition (2b) holds for the resulting graph and the process P instead of (2a). All other conditions were not changed. Therefore, the resulted graph G' is in the relation with the process $P \succ P_1 + P_2$ and we have a τ-transition:

$$G \xrightarrow{\tau} G', \text{ where } G^{\mathsf{rt}}_{P_1+P_2} \subseteq G, G^{\mathsf{rt}}_{P_1 \vee P_2} \subseteq G'$$

Thereby, the inference rule Summation being applied to the process $P_1 + P_2$ is equivalent to the application of the ccsSummation rule to a corresponding graph $G_{P_1+P_2}$.

The next step proceeds by the induction on the inner structure of P_1 or P_2. We have $P_1 + P_2 \xrightarrow{\tau} P'$, assume that P' is a subprocess of P_1. Therefore, according to the inference rule Summation we consider a process P_1. The next reasoning comes to the cases, when P_1 is either $P_{11} + P_{12}$, $P_{11} \mid P_{12}$, $\mu x.P_{11}$, $P[\mu x.P/x]$, $a.P_{11}$, or $a.P_{11} \mid \bar{a}.P_{12}$.

Case $P \succ P_1 \mid P_2$. Due to our assumption that $(P, G) \in \mathcal{R}_{CCS}$, G includes a corresponding graph for the process $P_1 \mid P_2$. Then according to (3), G includes either $G^{\mathsf{rt}}_{P_1 \wedge P_2}$ or $G^{\mathsf{rt}}_{P_1|P_2}$. In the first case the proof proceeds by the induction on the inner structure of P_1 or P_2. In the second case we proceed with arguing that the ccsComposition rule can be applied and that the resulted graph is still in the relation.

By Proposition 6.1.2 (Com) there exists a graph $G_{P_1|P_2}$, which is a corresponding for $P_1 \mid P_2$ and there are subgraphs $G_{P_1}, G_{P_2} \subseteq G_{P_1|P_2}$, which are corresponding graphs for the subprocesses P_1 and P_2, respectively. By (3a) and Remark 1 there exists a Current-node in G, which points to the subgraph $G_{P_1|P_2}$.

We can construct a match for the ccsComposition rule in $G^{\mathsf{rt}}_{P_1|P_2}$ graph (see Figure 6.21 and Table 6.1). In the resulting graph G', the Current-node v_{Curr} is deleted and two Current-nodes are created: v^1_{Curr} and v^2_{Curr} such that $current(v^1_{Curr}, v^1_{Proc})$ and $current(v^2_{Curr}, v^2_{Proc})$. The resulting graph G' includes $G^{\mathsf{rt}}_{P_1 \wedge P_2}$ (see Figure 6.14). The condition (3b) holds for G' and the process P instead of (3a). All other conditions of $\mathcal{R}_{CCS}$ were not changed. Therefore, G' is in the relation with the process $P = P_{P_1|P_2}$ and we have a τ-transition:

$$G \xrightarrow{\tau} G', \text{ where } G^{\mathsf{rt}}_{P_1|P_2} \subseteq G, G^{\mathsf{rt}}_{P_1 \wedge P_2} \subseteq G'$$

Thereby, the inference rule CompositionI being applied to the process $P_1 \mid P_2$ is equivalent to the application of the ccsComposition rule to a corresponding graph $G_{P_1|P_2}$.

The next step proceeds by the induction on the inner structures of P_1 and P_2. According to the inference rule Composition we consider both processes P_1 and P_2. The next reasoning comes to the cases, when the processes have one of the following graph structures: $P_{11} + P_{12}$, $P_{11} \mid P_{12}$, $\mu x.P_{11}$, x, $a.P_{11}$, or $a.P_{11} \mid \bar{a}.P_{12}$.

Cases $P \succ \mu x.P_1$ **and** x. Due to our assumption that $(P, G) \in \mathcal{R}_{CCS}$, G includes the corresponding graph for the process $\mu x.P_1$. Then according to (5), G includes $G^{\mathsf{rt}}_{\mu x.P_1}$.

By Proposition 6.1.2 (Rec) there exists a graph $G_{\mu x.P_1}$, which is a corresponding graph for P and there is a subgraph $G_{P_1} \subseteq G_{\mu x.P_1}$, which is a corresponding graph for the subprocess P_1. By (5) and Remark 1 there exists a Current-node in G, which points to the root node v_{Rec} of a subgraph $G_{\mu x.P_1}$, i.e. $G^{\mathsf{rt}}_{\mu x.P_1} \subseteq G$.

We can construct a match for the ccsVariableDeclaration rule in the graph $G^{\mathsf{rt}}_{\mu x.P_1}$(see Figure 6.22). In the resulting graph G', the Current-node

v_{Curr} is moved to the next Process-node such that $current(v^1_{Curr}, v_{Proc})$ and $subprocess(v_{Rec}, v_{Proc})$. The resulting graph G' consists of $G^{\mathsf{rt}}_{P_1}$, where the graph G_{P_1} is a corresponding process $P_1[\mu x.P_1/x]$. Due to Remark 2 and that the other conditions for the relation $\mathcal{R}_{CCS}$ did not change, we have $(P', G') \in \mathcal{R}_{CCS}$.
We have a τ-transition:

$$G \xrightarrow{\tau} G', \text{ where } G^{\mathsf{rt}}_{\mu x.P_1} \subseteq G, G^{\mathsf{rt}}_{P_1} \subseteq G'$$

If in the corresponding process $P = \mu x.P_1$ after the inference rule Recursion was applied, all occurences of x in the process P were replaced with P, then there is no variable x in P. However, the semantics of the corresponding CCS graph is different. There is a case, when a Current-node points to the Variable-node, i.e. we have a graph G^{rt}_x. We explain further why the inference rule Recursion is equivalent to the ccsVariableDeclaration and ccsRecursion rules.

The substitution part, i.e. $P_1[\mu x.P_1/x]$, happens, when a Current-node reaches a Variable-node. Then, the Current-node is moved to the root node of the graph G_{P_1}. For this, we apply the ccsRecursion rule. Therefore another τ-transition happens.

Thereby, the inference rule Recursion being applied to the process $P = \mu x.P_1$ is equivalent to the application of the ccsVariableDeclaration rule and the ccsRecursion rule to a corresponding graph G_P.

The next step proceeds by the induction on the inner structure of P_1. The next reasoning comes to the cases, when the process has one of the following graph structures:$P_{11} + P_{12}$, $P_{11} \mid P_{12}$, $\mu x.P_{11}$, x, $a.P_{11}$, or $a.P_{11} \mid \bar{a}.P_{12}$.

Case $P \succ a.P_1$. Due to our assumption $(P, G) \in \mathcal{R}_{CCS}$, G includes a corresponding graph for the process $a.P_1$. By Proposition 6.1.2 (Seq) $G_{a.P_1}$ is a corresponding graph for $a.P_1$ and a subgraph G_{P_1} is a corresponding graph for P_1. By (1) and Remark 1 there exists a Current-node in G such that $G^{\mathsf{rt}}_{a.P_1} \subseteq G$. Thereby, we can build a match for the ccsSequenceA rule (see Figure 6.17 and Table 6.1).

Let v_{Seq} be a root node of $G_{a.P_1}$ and v_{Proc} be a root node of G_{P_1}. Then in the resulting graph G' we have $current(v_{Curr}, v_{Proc})$. However, the ccsSequenceA rule has additional conditions on some other nodes that can be in match. These Mark and Current-nodes satisfy the following conditions:

$$\forall v_{Mark}, \tilde{v}_{Curr} : mark(v_{Mark}, v_{Curr}) \wedge mark(v_{Mark}, \tilde{v}_{Curr})$$

If such nodes exist, then the rule deletes these nodes, i.e. v_{Mark} and $\tilde{v}_{Curr}$. Further, we analyse how the deleted nodes affect the conditions of $\mathcal{R}_{CCS}$.

Note that a case when two or more Current-nodes are connected to a single Mark-node is possible only, when the ccsSummation rule was applied before and after that no ccsSequenceA and no ccsCoActionA rules were applied (since these rules remove v_{Curr} and $\tilde{v}_{Mark}$). Due to Proposition 6.1.1 the graph G has several subgraphs $G^{\mathsf{rt}}_{\bar{P}_1}$, $G^{\mathsf{rt}}_{\bar{P}_2}$, $G^{\mathsf{rt}}_{\bar{P}_3}$, ..., which correspond to one of the subprocesses from the first level of the process $P = \bar{P}_1 + \bar{P}_2 + \bar{P}_3 + \ldots$. We illustrtate the graph G schematicly with its emhasized subprocesses, where a root Process-node is connected with a Current-node. A common thing for all subgraphs is that all Current-nodes are connected with a single Mark-node (see the result in Figure 6.25).

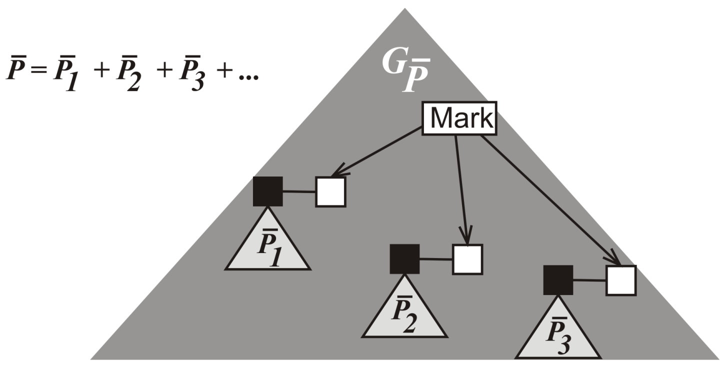


Figure 6.25: A schematically drawn CCS graph $G_{\bar{P}_1 \vee \bar{P}_2 \vee \ldots}$. Here, the black squares are Process-nodes, white squares are Current-nodes, triangles are the subgraphs $G^{\mathsf{rt}}_{\bar{P}_1}$, $G^{\mathsf{rt}}_{\bar{P}_2}$, $G^{\mathsf{rt}}_{\bar{P}_3}$, with a root node marked as a black square

So, we deal with a graph $G^{\mathsf{rt}}_{\bar{P}_1 \vee \bar{P}_2 \vee \ldots}$, where (let assume) $\bar{P}_1 = a.P_1$. It means that the original process is $P \succ a.P_1 + \bar{P}_2 + \bar{P}_3 + \ldots$ and it is in relation with the graph $G \supseteq G^{\mathsf{rt}}_{\bar{P}_1 \vee \bar{P}_2 \vee \bar{P}_3 \ldots}$. Then there is a transition in $Q_1(P)$:

$$P \xrightarrow{EventA} P', \text{ where } P \succ a.P_1 + \bar{P}_2 + \bar{P}_3 + \ldots P_1, P' \succ P_1$$

In the graph G we built a match for the ccsSequenceA rule. It we remember that $map_2(\text{ccsSequenceA}) = \text{EventA}$ we have a transition:

$$G \xrightarrow{EventA} G', \text{ where } G^{\mathsf{rt}}_{a.P_1 \vee \bar{P}_2 \vee \bar{P}_3 \ldots} \subseteq G, G_{P_1} \subseteq G'$$

Thereby, the inference rule Prefixing is equivalent to the application of the ccsSequenceA rule to a corresponding graph G_P. The resulted process and the resulted graph are corresponding. Additionally, the Current-nodes point to the root element. Then according to Remark 2 and the fact that all the other conditions of $\mathcal{R}_{CCS}$ did not change, the $(P_1, G_{P_1}) \in \mathcal{R}_{CCS}$.

Case $P \succ a.P_1 \mid \bar{a}.P_2$ is similar to the previous case. Here, we assume that the CompositionII inference rule was applied to the process $a.P_1 \,|\, \bar{a}.P_2$ during the justification process (for sure the Prefixing rule was applied twice at the ende). We want to show that it is equivalent to the application of the ccsCoActionA rule.

Due to our assumption $(P, G) \in \mathcal{R}_{CCS}$, G includes a corresponding graph for the process $a.P_1 \,|\, \bar{a}.P_2$. By Proposition 6.1.2 (Com) and (Seq) $G_{a.P_1|\bar{a}.P_2}$ is a corresponding graph for $a.P_1 \,|\, \bar{a}.P_2$ and subgraphs $G_{a.P_1}$, $G_{\bar{a}.P_2}$, G_{P_1} and G_{P_2} are corresponding graphs for $a.P_1$, $\bar{a}.P_2$, P_1 and P_2, respectively. By (3) and Remark 1 there exists a Current-node in G such that $G^{\mathsf{rt}}_{a.P_1|\bar{a}.P_2} \subseteq G$. Thereby, we can build a match for the ccsCoActionA rule (see Figure 6.19 and Table 6.1).

Let v^1_{Seq} and v^2_{Seq} be root nodes of $G_{a.P_1}$ and $G_{\bar{a}.P_2}$, respectively; v^1_{Proc} and v^2_{Proc} be root nodes of G_{P_1} and G_{P_2}, respectively. Then in the resulting graph G' we have $current(v^1_{Curr}, v^1_{Proc})$ and $current(v^2_{Curr}, v^2_{Proc})$. However, the ccsCoActionA rule has additional conditions on some other nodes that can be in match. These Mark and Current-nodes satisfy the following conditions:

$$\forall v_{Mark}, \tilde{v}_{Curr} : mark(v_{Mark}, v^1_{Curr}) \wedge mark(v_{Mark}, \tilde{v}_{Curr}) \vee$$

or

$$\forall v_{Mark}, \tilde{v}_{Curr} : mark(v_{Mark}, v^2_{Curr}) \wedge mark(v_{Mark}, \tilde{v}_{Curr})$$

If such nodes exist, then the rule deletes these nodes, i.e. v_{Mark} and $\tilde{v}_{Curr}$. Further, we analyse how the deleted nodes affect the conditions of $\mathcal{R}_{CCS}$. The existence of such nodes means that the ccsSummation rule was applied before to the graph G and then no ccsSequenceA or ccsCoActionA rules were applied after. Due to Proposition 6.1.1 we deal with a graph of a type $G^{\mathsf{rt}}_{(P_1+\bar{P}_1)\wedge(P_2+\bar{P}_2)+\bar{P}_3}$. If we apply the ccsCoActionA rule and remember that $map(\mathsf{ccsCoActionA}) = \mathsf{EventTau}$, we have a transition:

$$G \xrightarrow{EventTau} G', \text{ where } G^{\mathsf{rt}}_{(P_1+\bar{P}_1)\wedge(P_2+\bar{P}_2)+\bar{P}_3} \subseteq G, G^{\mathsf{rt}}_{P_1\wedge P_2} \subseteq G'$$

in the LTS $Q_1(P)$, we have a transition:

$$P \xrightarrow{\mathsf{EventTau}} P', \text{ where } P \succ a.P_1 \mid \bar{a}.P_2 + \bar{P}_2 + \bar{P}_3 + \ldots, P' \succ P_1 \mid P_2.$$

The resulting graph $G^{\mathsf{rt}}_{P_1\wedge P_2}$ is in the relation with the process $P_1 \mid P_2$ due to (3b). Thereby, the inference rule CompositionII is equivalent to the application of the ccsCoActionA rule to a corresponding graph G_P. The resulted process and the resulted graph are corresponding. Additionally, the Current-nodes point to the root elements. Then according to Remark 2 and the fact that the rest conditions of $\mathcal{R}_{CCS}$ did not change, the $(P', G'_{P'}) \in \mathcal{R}_{CCS}$.

For the **reverse direction** we assume again that some process P and a corresponding graph G are in the relation $\mathcal{R}_{CCS}$. We prove that for any transition $G \xrightarrow{r} G'$, there is some process P' such that $P \overset{\widehat{r}}{\Longrightarrow} P'$ with $(P', G') \in \mathcal{R}_{CCS}$.

$r = \mathsf{ccsSummation}$, then we have an invisible transition on the CCS graph-side. The transition $G \xrightarrow{\mathsf{ccsSummation}} G'$ means that there exists a subgraph $G^{\mathsf{rt}}_{P_1+P_2} \subseteq G$.

Due to Proposition 6.1.3 (Sum) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding subprocess of the first or second level $P_1 + P_2 \prec P$ for $G^{\mathsf{rt}}_{P_1+P_2}$ (see Figure 6.20 and Table 6.1). In the resulting graph G', there is a subgraph $G^{\mathsf{rt}}_{P_1 \vee P_2} \subseteq G'$. Therefore, the condition (2b) holds instead of (2a) for the resulting graph G' and the process P. All the other conditions did not change. Thus (P, G') is in $\mathcal{R}_{CCS}$ and $P \overset{\widehat{\tau}}{\Longrightarrow} P$.

$r =$ ccsComposition, then we have an invisible transition on the CCS graph-side. The transition $G \xrightarrow{\mathsf{ccsComposition}} G'$ means that there exists a subgraph $G^{\mathsf{rt}}_{P_1|P_2} \subseteq G$. Due to Proposition 6.1.3 (Com) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding subprocess of the first or second level $P_1 \mid P_2 \prec P$ for $G^{\mathsf{rt}}_{P_1|P_2}$ (see Figure 6.21 and Table 6.1). In the resulting graph G', there is a subgraph $G^{\mathsf{rt}}_{P_1 \wedge P_2} \subseteq G'$. Therefore, the condition (2b) holds instead of (2a) for the resulting graph G' and the process P. All the other conditions did not change. Thus (P, G') is in $\mathcal{R}_{CCS}$ and $P \overset{\widehat{\tau}}{\Longrightarrow} P$.

$r =$ ccsVariableDeclaration, then we have an invisible transition on the CCS graph-side and on the CCS process side. The transition $G \xrightarrow{\mathsf{ccsVariableDeclaration}} G'$ means that there exists a subgraph $G^{\mathsf{rt}}_{\mu x.P_1} \subseteq G$. Due to Proposition 6.1.3 (Rec) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding subprocess of the first or second level $\mu x.P_1 \prec P$ for $G^{\mathsf{rt}}_{\mu x.P_1}$ (see Figure 6.22 and Table 6.1). It means that we can apply the inference rules Summation, CompositionI or Recursion, which do not lead to a completed transition in the LTS, but simplify the process P to a process $P_1[\mu x.P_1/x]$, which corresponds to a graph G_{P_1} . Note that in the resulting graph G', there is a subgraph $G^{\mathsf{rt}}_{P_1} \subseteq G'$. Therefore, the condition (5b) holds for the resulting graph G'_{P_1} and the process P_1. The proof that all the other conditions hold is by induction on the inner structure of the process P_1. Thus $(P_1[\mu x.P_1/x], G'_{P_1})$ is in $\mathcal{R}_{CCS}$ and $P \overset{\widehat{\tau}}{\Longrightarrow} P_1$.

$r =$ ccsRecursion, then we have an invisible transition on the CCS graph-side. The transition $G \xrightarrow{\mathsf{ccsRecursion}} G'$ means that there exists a subgraph $G^{\mathsf{rt}}_{x} \subseteq G$. Due to Proposition 6.1.3 (Rec) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding subprocess of the first or second level $x \prec P$ for a graph G^{rt}_{x} (see Figure 6.23 and Table 6.1). Since we consider CCS processes, where x is always a subgraph of some process $\mu x.P_1$, then there exists a process $P_1[\mu x P/x]$, which corresponds to a graph G_{P_1}. The graph G_x is a subgraph of G_{P_1}. The ccsRecursion rule moves a Current-node to the graph G_{P_1}. In the resulting graph G', there is a subgraph $G^{\mathsf{rt}}_{P_1} \subseteq G'$. Therefore, conditions (5b) holds for the resulting graph G' and the process $P[\mu x P/x]$. The proof that all the other conditions hold is by induction on the inner structure of the process P_1. Thus (P, G') is in $\mathcal{R}_{CCS}$ and $P \overset{\widehat{\tau}}{\Longrightarrow} P$.

$r = \mathsf{ccsSequenceA}$, then there exists a subgraph $G^{\mathsf{rt}}_{a.P_1} \subseteq G$ and $G^{\mathsf{rt}}_{P_1} \subseteq G'$. Due to Proposition 6.1.3 (Seq) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding process of the first or second level $a.P_1 \prec P$ for a graph $G^{\mathsf{rt}}_{a.P_1}$ (see Figure 6.17 and Table 6.1). However, the applied ccsSequenceA rule can also delete additional Current- and Mark-nodes $\tilde{v}_{Curr}$, v_{Mark} such that if v_{Seq} is a root node of a graph $G^{\mathsf{rt}}_{a.P_1}$ and $v_{Curr} : current(v_{Seq}, v_{Curr})$ then

$$\forall \tilde{v}^i_{Curr}, v_{Mark} : mark(v_{Mark}, v_{Curr}) \wedge mark(v_{Mark}, \tilde{v}^i_{Curr})$$

If such nodes exist, then a graph G according to Proposition 6.1.1 has a subgraph $G^{\mathsf{rt}}_{a.P_1 \vee \bar{P}_1 \vee \bar{P}_2 \vee \ldots}$, where v^i_{Proc} is a root Process-node of some graphs G_{P_i}, i.e. $current(\tilde{v}^i_{Curr}, v^i_{Proc})$. Due to Proposition 6.1.3 (Sum) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a subprocess of the first or second level $a.P_1 + \bar{P}_1 + \bar{P}_2 + \ldots$. It is also not excluded that there are other subprocesses of the first and second levels of P. However, we can apply the inference rules Summation, CompositionI and Recursion, leading to the subprocess $a.P_1$ and not completed transition in the LTS. Only at the end the inference rule Prefixing can be applied, that leads to a transition $P \xrightarrow{a} P_1$ being justified. The resulted process P' corresponds to the graph G'_{P_1} and we know a Current-node points to the root of this graph, i.e. $G^{\mathsf{rt}}_{P_1}$. Therefore, we have $(P', G') \in \mathcal{R}_{CCS}$.

$r = \mathsf{ccsCoActionA}$, then there exists a subgraph $G^{\mathsf{rt}}_{a.P_1|\bar{a}.P_2} \subseteq G$ and $G^{\mathsf{rt}}_{P_1|P_2} \subseteq G'$. Due to Proposition 6.1.3 (Com) and (Seq) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a corresponding process of the first or second level $a.P_1 \mid \bar{a}.P_2 \prec P$ for $G^{\mathsf{rt}}_{a.P_1}$ (see Figure 6.19 and Table 6.1). However, the applied ccsCoActionA rule can also delete additional Current- and Mark-nodes such that if v^1_{Seq} and v^2_{Seq} are root nodes of graphs $G^{\mathsf{rt}}_{a.P_1}$ and $G^{\mathsf{rt}}_{\bar{a}.P_1}$, respectively, $v^1_{Curr} : current(v^1_{Curr}, v^1_{Seq})$ and $v^2_{Curr} : current(v^2_{Curr}, v^2_{Seq})$ then

$$\forall \tilde{v^1}_{Curr}, v^1_{Mark} : mark(v^1_{Mark}, v^1_{Curr}) \wedge mark(v^1_{Mark}, \tilde{v^1}_{Curr})$$

and

$$\forall \tilde{v^2}_{Curr}, v^2_{Mark} : mark(v^2_{Mark}, v^2_{Curr}) \wedge mark(v^2_{Mark}, \tilde{v^2}_{Curr})$$

If such nodes exist, then a graph G according to Proposition 6.1.1 has a subgraph $G^{\mathsf{rt}}_{(a.P_1 \vee \bar{P}_1 \vee \ldots)|(\bar{a}.P_2 \vee \bar{P}_2 \vee \ldots) \vee \bar{P}_1 \vee \bar{P}_2 \vee \ldots}$. Due to Proposition 6.1.3 (Sum) and our assumption that $(P, G) \in \mathcal{R}_{CCS}$, there exists a subprocess of the first level $(a.P_1 + \bar{P}_1 + \ldots) \mid (\bar{a}.P_2 + \bar{P}_2 + \ldots) + \bar{P}_1 + \bar{P}_2 + \ldots$. It is also not excluded that there are other subprocesses on the first and second levels of P. However, we can apply the inference rules Summation, CompositionI and Recursion, leading to the subprocess $(a.P_1 + \bar{P}_1 + \ldots) \mid (\bar{a}.P_2 + \bar{P}_2 + \ldots)$. Only then the rule CompositionII, and again Summation. At the end the inference rule Prefixing can be applied, that leads to a transition $P \xrightarrow{\tau} P_1$ being justified. Since $\tau \in \mathit{Act}$, then we have a transition $P \xrightarrow{\mathsf{EventTau}} P_1$. The resulted process P' corresponds to the graph G'_{P_1} and we know a $\mathsf{Current}$-node points to the root of this graph, i.e. $G^{\mathsf{rt}}_{P_1}$. Therefore, we have $(P', G') \in \mathcal{R}_{CCS}$.

□

6.2 Petri Nets (Steps 1-2)

Recall that in Section 6.1 we defined the CCS language, which defines source graphs for model transformation. In this section, we define our target language (Steps 1-2 of our method). To be more precise, we define the syntax and behavioural semantics of the Petri nets language by means of graph transformations.

Petri nets are used in practice for description of communication protocols, which ensure reliable transmission between hosts. Petri nets describe how packages could be sent and received. A Petri net consists of a set of *places* and a set of *transitions*, the latter are defined over the places in such way that each transition has at least one *input* place and one *output* place. *Transitions* represent transmission of some data. Places are used to represent states of a modelled system. A *marking* function is defined for each place. If the value of this function is greater than zero, it means that the place contains a package. The transmission of data is defined by changing the markings on each step [JK09].

The original definition of Petri nets is given within set theory and the behavioural semantics is defined by the use of a marking function. In this section we specify the Petri nets language according to the requirements of our method (Figure 5.2), i.e. by means of graph transformations (see Figure 6.26).

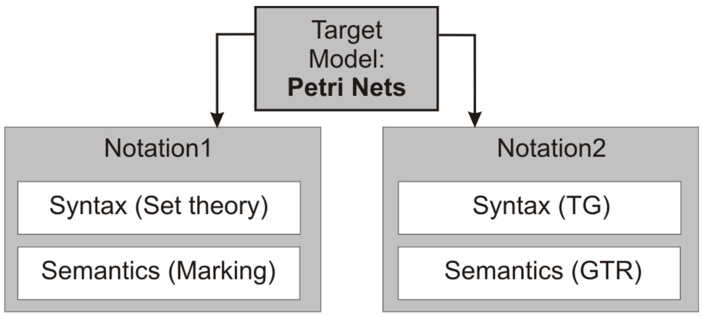


Figure 6.26: The goal of this section is to define the Petri nets language with a Type Graph (TG) and Graph Transformation Rules (GTRs)

We start with a basic definition of Petri nets, from which we construct a type graph. Then, we specify behavioural semantics by means of graph transformations.

6.2.1 Syntax

We consider a standard definition of Petri nets [Kot78].

Definition 26 (Petri nets)**.** A *Petri net* is a tuple $N = \langle S, T, I \rangle$, where S is a nonempty finite set of places, $T \subseteq N_+^S \times Act \times N_+^S$ is a finite set of transitions, N_+^S denotes a multiset of places including at least one element, Act is a set of labels, $I \subseteq S$ is a set of initial places.

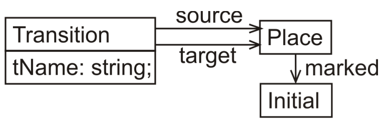


Figure 6.27: Type graph T^{st}_{PN} for Petri nets

To implement such a structure with a type graph, we do the following. For each set we define a separate node: a Place-node for the set S, a Transition-node for the set T, an Initial-node for the set I. Since transitions are defined as a product over the set S and set of labels Act, we specify an attribute for a Transition-node, which value corresponds to an element from Act, and we use the edges of types source and target to implement the mapping function $\times$ of the product. To specify that the set I is a subset of S in the type graph, we use an edge of a type Initial. We assume that each Transition-node is always connected to at least one Place-node with a source edge and to at least one Place-node with a target-edge. Therefore, each Transition-node has at least one input Place-node and at least one output Place-node. If a Place-node is connected to an Initial-node, it means that the place is from the set I. We present a type graph of Petri nets in Figure 6.27.

6.2.2 Semantics

The behavioural semantics of Petri nets is originally defined by the use of a *marking* function $M : S \to N \cup \{0\}$, which marks each place as a natural number or zero. Each transition is defined as a triple (preset, label, postset). The preset of a transition is denoted by $\bullet t$, the label by $l(t) \in Act$ and the postset by $t\bullet$. A preset and a postset stand for the number of input and output places, respectively. Then, the behaviour of a Petri net is defined by changing values of a marking function. The change is accomplished with two rules, which firstly check if a transition is enabled by M and then define a new marking for the input and output states.

Definition 27 (Behavioural semantics of Petri nets)**.** Let $N = \langle S, T, I\rangle$ be a Petri net, M, M' are markings. Then a step from M to M' occurs for a transition $t \in T$ iff

1. for all $s \in S$, $M(s) \geq \bullet t(s)$ (the condition which enables the transition t),
2. for all $s \in S$, $M'(s) = M(s) - \bullet t(s) + t \bullet (s)$ (the marking is changed from M to M').

To implement the behavioural semantics of Petri nets, we use a Token-node to denote a marking function M. Zero or more Token-nodes can be connected with a Place-node. See the run-time graph T^{rt}_{PN} in Figure 6.28, which is the type graph T^{st}_{PN} extended with a Token-node. We consider the initial marking of Place-nodes. For this we define the graph transformation rule, called pnInitial, which creates exactly one Token-node for each Place-node which is labelled as initial (see Figure 6.29).

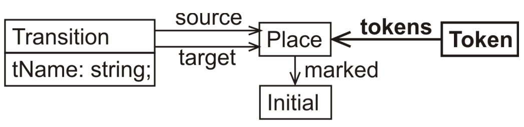


Figure 6.28: Run-time graph T^{rt}_{PN} for Petri nets

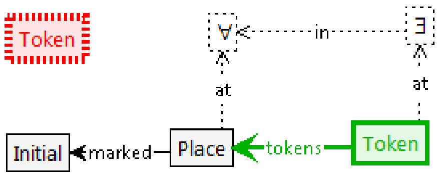


Figure 6.29: The pnInitial rule (creates exactly one Token-node for each Place-node which is labelled as initial)

In order to implement conditions (1)-(2) from Definition 27, we define a graph transformation rule pnMoveToken. The pnMoveToken rule checks (1), i.e. if a Transition-node is enabled. To be more precise, each input Place-node must be connected with a Token-node. If a match is found, the pnMoveToken rule performs the change of a marking. The rule removes one Token-node for every input Place-node and creates exactly one Token-node for every output Place-node (see Figure 6.30).

We also want to know the attribute value of a Transition-node the rule was applied to. Therefore, we consider the rule from Figure 6.30 as a template, which is extended each time for each attribute value of a Transition-node. An example of the extension is presented in Figure 6.31 and is called pnMoveTokenBig rule (an attribute value is always added to pnMoveToken). In the pnMoveTokenBig rule the Transition-node is extended with an attribute value *big*. Throughout the remainder of this thesis, pnMoveToken stands for any of the extended rules from this template.

We define a semantic rule system as a partial mapping $\mathcal{RS}_{PN} : \mathsf{Sym}_{PN} \rightharpoonup \mathsf{Rule}_{PN}$, where Rule_{PN} is a set of graph transformation rules from Figures 6.29 and 6.30, $\mathsf{Sym}_{PN} = \{\mathsf{pnInitial}, \mathsf{pnMoveToken}\}$ is a set of semantic rule names for the Petri nets graphs. Note that here, pnMoveToken stands for a set, which consists of all possible extensions from the respective template for each label from *Act*.

For later, we use additional notation for Place-nodes:

$initial(v_{Place})$, if there exists an Initial-node v_{Init} and a Place-node v_{Place} such that $current(v_{Init}, v_{Place})$.

$token(v_{Place})$, if there exists a Token-node v_{Tok} and a Place-node v_{Place} such that $tokens(v_{Tok}, v_{Place})$.

We use $\mathcal{G}^{\mathsf{st}}_{PN}$ and $\mathcal{G}^{\mathsf{rt}}_{PN}$ to denote the set of all well-formed syntactic and semantics Petri nets graphs, respectively.

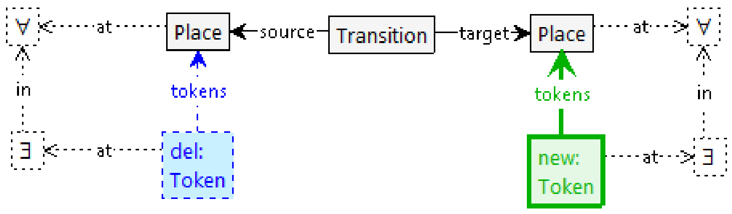


Figure 6.30: A template for the pnMoveToken rule (removes one Token-node from each input Place-node, creates one Token-node for each output Place-node)

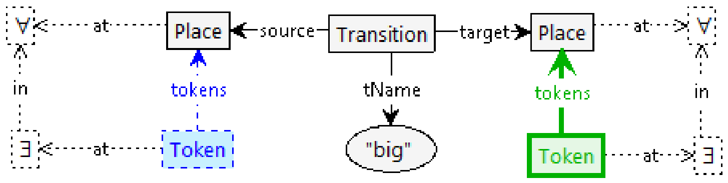


Figure 6.31: The pnMoveTokenBig rule (removes one Token-node from each input Place-node, creates one Token-node for each output Place-node) is an extension of the pnMoveToken template with the attribute node for the Transition-node with the value *big*

6.3 Mapping over the Rule Systems (Step 3)

Our objective is to compare two LTSs: the first LTS is generated for a graph G_{CCS} of the CCS language (denoted later as $Q(G_{CCS})$), the second LTS is generated for a graph G_{PN} of the Petri net language (denoted later as $Q(G_{PN})$). Graphs G_{CCS} and G_{PN} are received as a result of a model transformation $MT_{CCS2PN} \subseteq \mathcal{G}^{\mathsf{st}}_{CCS} \times \mathcal{G}^{\mathsf{st}}_{PN}$ (defined formally in Section 6.4) translating CCS-graphs into Petri nets graphs. We aim at proving this model transformation to be behaviour preserving, in the sense that the LTSs of source and target models are always weak bisimilar. However, there is an obvious problem: the LTSs do not have the same labels. Therefore, in this section we define a mapping on the labels (i.e. rule names) to a common set of names.

Although, $dom(\mathcal{RS}_{CCS}) \cap dom(\mathcal{RS}_{PN}) = \emptyset$, by providing a short analysis, we can find out, which rules correspond to each other. An Event-node and a Transition-node have obviously the same meaning, because they originally denote a process that takes place. Therefore, we map the rules, which keep control over the dynamic elements, when the latter flow through the nodes which have the same meaning. These rules are the ccsSequenceA rule and the pnMoveTokenA rule, the ccsCoActionA rule and the pnMoveTokenTauA rule. The ccsInitial and pnInitial rules denote a creation of dynamic elements in the syntactic graph, therefore we map these rules to each other too. The rest rules do not provide any changes over Event- and Transition-nodes, thus, they do not have a counterpart and are seen as internal steps. These observations give rise to the following non-trivial mapping (i.e. not all rules are mapped to an invisible step) defined on the labels of LTSs to a common set of names $\mathsf{Sym} = \{\mathsf{Action}, \mathsf{Initial}, \tau\}$:

$$
\begin{array}{lll}
map_{CCS}\colon & \forall a \in Act \backslash \tau & \mathsf{ccsSequenceA} \mapsto \mathsf{ActionA}, \\
& \forall a \in Act \backslash \tau & \mathsf{ccsCoActionA} \mapsto \mathsf{ActionTauA}, \\
& \mathsf{ccsInitial} \mapsto \mathsf{Initial}, & \\
& \mathsf{ccsSummation} \mapsto \tau, & \\
& \mathsf{ccsComposition} \mapsto \tau, & \\
& \mathsf{ccsRecursion} \mapsto \tau, & \\
& \mathsf{ccsVariableDeclaration} \mapsto \tau. & \\
map_{PN}\colon & \mathsf{pnInitial} \mapsto \mathsf{Initial}, & \\
& \forall a \in Act \backslash \tau & \mathsf{pnMoveTokenA} \mapsto \mathsf{ActionA}, \\
& \forall a \in Act \backslash \tau & \mathsf{pnMoveTokenTauA} \mapsto \mathsf{ActionTauA}.
\end{array}
$$

There are two important notes:

1. Here, it is implied that for every rule ccsSequenceA received as a result of the extension of the ccsSequence template there exists a corresponding rule pnMoveTokenA, recieved as a result of the extension of the pnMoveToken template (the ccsSequenceA rule is applied to the Event-node with the attribute value a, the pnMoveTokenA rule is applied to the Transition-node with the attribute value a). Therefore, the mapping is defined for every attribute value a (A).
2. The pnMoveTokenTauA rule implements a parallel execution of the event a and its co-event $\bar{a}$. The meaning of the parallel execution in Petri nets is explained in the next section.

The mapping map_{CCS} and map_{PN} are defined on the LTSs, by mapping only the label sets to Sym. Then, our objective could be formalized as the following statement to be shown:

$$map_{CCS}(Q(G_{CCS})) \approx map_{PN}(Q(G_{PN})) \tag{6.1}$$

here $G_{CCS} \in \mathcal{G}^{\mathsf{st}}_{CCS}$, $G_{PN} \in \mathcal{G}^{\mathsf{st}}_{PN}$ with $MT_{CCS2PN}(G_{CCS}, G_{PN}) \subseteq \mathcal{G}^{\mathsf{st}}_{CCS} \times \mathcal{G}^{\mathsf{st}}_{PN}$. The mappings $map_{CCS}\colon dom(\mathcal{RS}_{CCS}) \to \mathsf{Sym}$ and $map_{PN}\colon dom(\mathcal{RS}_{PN}) \to \mathsf{Sym}$ are non-trivial functions, i.e. they do not map all names of semantic rules to one label. $\approx$ denotes weak bisimulation.

6.4 Model Transformation Specification (Step 4)

In this section we define a model transformation $MT_{CCS2PN} \subseteq \mathcal{G}^{\mathsf{st}}_{CCS} \times \mathcal{G}^{\mathsf{st}}_{PN}$ translating CCS graphs into Petri net graphs. The definition includes the explanation of the Triple Graph Grammar (TGG) technique that we use, the idea of mapping, graph transformations itself. We proceed as follows. The idea of the TGG technique is already introduced in Chapter 3, we explain how we use it for our case in Subsection 6.4.1. The mapping within the model transformation is formally defined in Subsection 6.4.2. The graph transformation system for the model transformation between CCS and Petri nets is specified in Subsection 6.4.3.

6.4.1 TGG Model Transformation

In order to show the correctness of a model transformation, it is crucial to keep correspondences between nodes of source and target models during model transformation. It allows later to reason about properties of transformed models. Therefore, we use the TGG technique.

The main idea of the TGG technique is that graphs are separated into three subgraphs, each being typed over its own type graph. One graph is normally typed over the source type graph and another is typed over the target type graph. Two of these subgraphs evolve simultaneously while the third keeps correspondences between them. In our case we have the type graph T^{st}_{CCS} for the CCS syntax and the type graph T^{st}_{PN} for the Petri nets syntax, which are conjoined with one new correspondence node. The correspondence node is connected to the nodes, which have the same meaning, i.e. Event-nodes and Transition-nodes. This correspondence node represents the third graph in TGG (see Figure 6.32).

As opposed to traditional transformation where the source model is given and then the source model is replaced by the target model, TGG transformations build the models simultaneously, matching each part of the source model to the target one. This allows to keep correspondences between transformed elements and to prove certain properties of the corresponding graphs.

In our model transformation we use *Building Blocks* (BBs) – additional elements, – that have similar meaning to CCS subprocesses and subnets in the Petri nets language. BBs evolve into the full process or net by a partial substitution on each

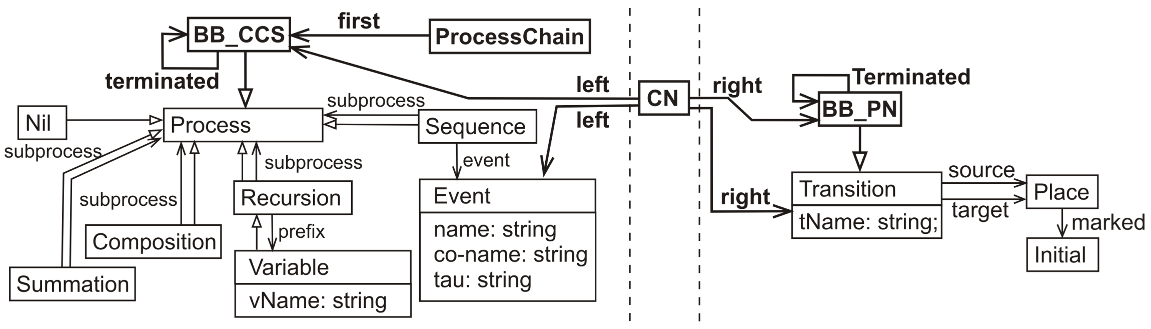


Figure 6.32: Type graph $T^{\mathsf{st}}_{CCS} \times T_{CN} \times T^{\mathsf{st}}_{PN}$ for TGG transformation

step of a certain construction. Full substitution of BB happens on the final steps when we transform CCS BBs into CCS events and Petri net BBs into transitions. As the deletion of elements in the original TGG is prohibited, the substitution process is represented by adding a special marker *Terminated*, that could also mean that BB is no longer active. Each CCS BB is connected with a Petri net BB via a correspondence node. This connection extends to transitions of Petri nets and events of CCS language, i.e. to Transition- and Event-nodes.

Since we use the new nodes during the model transformation, we extend our type graphs. We introduce a BB_CCS-node which has all properties of the Process-node (see Figure 6.32). A BB_CCS-node has a self Terminated-edge. ProcessChain is another auxiliary node that helps to build a root node of the CCS tree. The Petri net type graph T^{st}_{PN} is extended with a BB_PN-node, which inherits a Transition-node. Similar to a BB_CCS-node, a BB_PN-node has a self Terminated-edge. The type graph T_{CN} consists of only one CN-node or a *corresponding* node which connects either BB_CCS- and BB_PN-nodes or Event- and Transition-nodes.

So, the TGG rules MT_{CCS2PN} build combined CCS and Petri nets graphs. We let $\mathcal{G}^{\mathsf{st}}_{CCS2PN}$ to denote the set of graphs obtained by applying the TGG rules MT_{CCS2PN} on an empty start graph. To receive the final translation, we need to project the graph $G_{CCS2PN} \in \mathcal{G}^{\mathsf{st}}_{CCS2PN}$ onto the CCS and the Petri nets type graphs. We use the definition of projection defined in Chapter 3, Section 3.1 to specify the model transformations MT_{CCS2PN}: Given a CCS graph G_{CCS} and a Petri net graph G_{PN}, we have $MT_{CCS2PN}(G_{CCS}, G_{PN})$ exactly if there is some $G_{CCS2PN} \in \mathcal{G}^{\mathsf{st}}_{CCS2PN}$ such that $G_{CCS} = \pi_{T^{\mathsf{st}}_{CCS}}(G_{CCS2PN})$ and $G_{PN} = \pi_{T^{\mathsf{st}}_{PN}}(G_{CCS2PN})$.

6.4.2 Mapping of Well-Formed CCS Graphs to Petri Nets

The model transformation of CCS into Petri nets is based on a mapping of syntactic elements from the type graphs. The basic mapping is based on the assumption that an Event-node and a Transition-node have the same meaning. Then, the mapping is defined for each well-formed construction of CCS graphs (see Figure 6.9).

To define a corresponding Petri net construction we use the extended Petri nets type graph with BBs nodes (see Figure 6.33), which was already explained in the previous subsection.

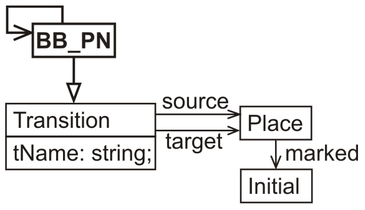


Figure 6.33: Extended type graph for Petri nets

We inductively define further patterns of Petri nets graphs. For this, we provide a mapping between each defined case of a well-formed CCS graph to a Petri net pattern except recursion, which is defined separately. We start with a mapping of a CCS BB to a Petri net BB. The latter consists of two Place-nodes and a node of the type BB_PN that has exactly two incoming edges: one connecting it to the Arc-node by source-edge and another connecting it to the Arc-node by a target-edge (see Figure 6.34 in the top left corner). Then we define four cases (see Figure 6.34), each of them is labelled with a letter, which corresponds to a label of a well-formed CCS graph from Figure 6.9:

- **Case A:** *nil* process is mapped to an empty Petri net, which is represented as a Place-node (see Figure 6.34A).
- **Case B:** $P_1 + P_2$ is mapped to a Petri net, which represents a choice. Such structure consists of a Place-node followed by two BBs (see Figure 6.34B).
- **Case C:** $P_1 \mid P_2$ is mapped to a Petri net, which represents a parallel composition. Such structure consists of two BBs (see Figure 6.34C).
- **Case D:** $a.P_1$ is mapped to a Petri net, which represents a sequence. Such structure consists of a Place-node, followed by a Transition-node, another Place-node and finally a BB (see Figure 6.34D).

Cases A-D define a special class of Petri nets, called *synchronisation free* Petri nets [Val94], i.e. each transition has at most one incoming edge. Such class of nets represents CCS processes with unboundedly growing parallelism.

We extend the class of synchronisation free Petri nets with the definition of co-event (i.e. event $\bar{a} \in \bar{\Delta}$). As we mentioned earlier each Event-node corresponds to a Transition-node. However, if there are two Transition-nodes in Petri net, which represent events a and $\bar{a}$, then there is a special transition, which has input and output nodes of the transitions a and $\bar{a}$ (see Figure 6.35).

We introduce a new graph transformation rule pnMoveTokenTauA (see Figure 6.36), which moves a token through the transition with an attribute value τ.

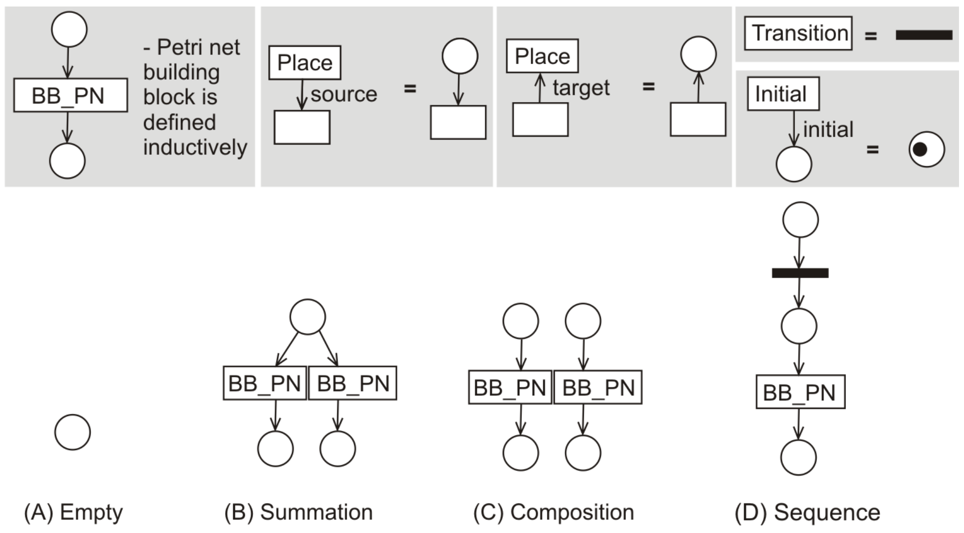


Figure 6.34: Definition of well-formed Petri nets graphs

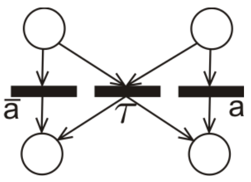


Figure 6.35: Definition of a co-event in Petri nets

We proceed with the definition of recursion for the defined Petri nets. The recursion is an interesting case, which requires additional explanation. We have two cases for a definition of recursion. The reason for this is that composition in Petri nets is represented differently, depending if a net forms a *connected net* or two *separate nets* (see case C from Figure 6.34). Therefore, there are two cases for recursion. In the first case, recursion is defined by a mapping of a place to a certain place, where recursive loop starts (i.e a process $a.\mu x.b.x$). In the second case, there is a need to forward all target edges to initial places (i.e a process $\mu x.(a.x \mid b.nil)$).

Definition 28 (Recursion)**.** Let $N = \langle S, T, I \rangle$ be a Petri net, then the recursion in N is a subnet, which is a loop Petri net, starting in place $s \in S$ and proceeding with a connected subnet until transition $t = (N_+^{s'}, l(t), N_+^{s''})$, where $l(t)$ is a label for t. The recursion is defined by changing the set of output places for each transition $\tilde{t}$, which has an output place from set $N_+^{s''}$:

$$\tilde{t} := (N_+^{\tilde{s'}}, l(\tilde{t}), N_+^{\tilde{s''}} \cup N_+^{s})$$

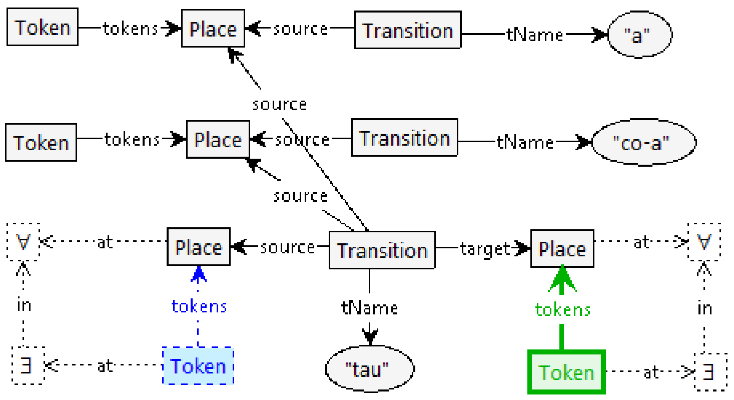


Figure 6.36: The pnMoveTokenTauA rule

Definition 29 (RecursionI)**.** Let $N = \langle S, T, I\rangle$ be a Petri net, then the recursion followed by transition $t = (N_+^{s'}, l(t), N_+^{s''})$, where $l(t)$ is a label for t, is defined by changing the set of output places for each transition $\tilde{t}$, which has an output place from set $N_+^{s''}$:

$$\tilde{t} := (N_+^{\tilde{s}'}, l(\tilde{t}), N_+^{\tilde{s}''} \cup I)$$

6.4.3 Graph Transformation System

We had to implement the mapping described in the previous subsection. For this, we defined graph transformation rules, which specify our transformation system MT_{CCS2PN}. The rules mainly carry out four tasks: (1) creation of a skeleton for transformation in an empty graph, (2) implementation of a mapping between well-formed patterns described earlier (see Figure 6.34), (3) creation of Event-nodes from sets $\{\tau\}$ and $\bar{\Delta}$ and corresponding transitions in Petri nets, and finally (4) implementation of recursion.

We need one rule for performing task (1). The next four rules implement task (2). These rules evolve BB by its “replacement” with patterns for well-formed CCS graph and Petri nets graphs. The next two rules carry out task (3). Recursion, i.e. task (4), is defined with three rules, one of which creates process $\mu x.P$, and the other two implement Definitions 28–29. Table 6.4.3 summarizes these rules and briefly states their task within the transformation process.

Note. Since the task of each rule is to build a well-formed construction of a CCS graph and a Petri net graph (see Figures 6.9 and 6.34) we add some additional information. In Table 6.4.3 we mention a letter of the CCS well-formed graph case (see Figure 6.9) and a letter of the Petri nets well-formed graph case (see Figure 6.34) in the brackets. For example, (A $\leftrightarrow$ A) means that a rule builds the structure

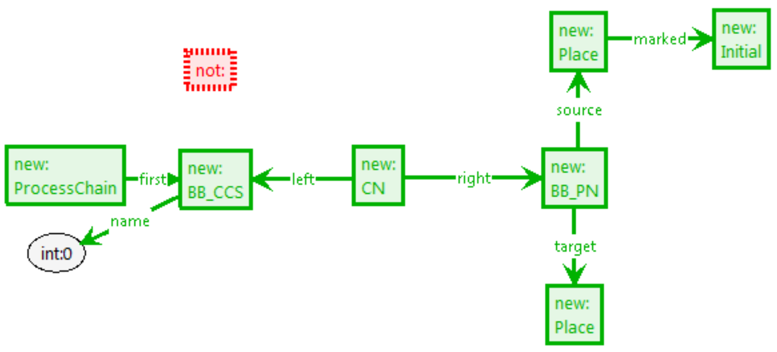


Figure 6.37: The tggInitial rule (creates a skeleton, i.e. the first BBs: a BB_CCS-node and a BB_PN-node)

of the process *nil*, which is case A in Figure 6.9, and a structure of the empty Petri net, which is case A in Figure 6.34.

In the following, we discuss each rule in detail.

The tggInitial rule

The tggInitial rule (see Figure 6.37) is executed always at first. It creates in an empty graph a skeleton for the transformation, i.e. two corresponding BBs: a BB_CCS-node and a BB_PN-node. In addition, the rule creates an auxiliary ProcessChain-node in a CCS graph. In a Petri nets graph, the tggInitial rule creates a Place-node connected with an Initial-node, which is an input place for a newly created BB_PN-node, and an outcome Place-node. Finally, a corresponding CN-node is created which joins the BB_CCS and BB_PN-node.

The tggEmpty rule

The tggEmpty rule turns two corresponding BBs into a structure, which corresponds to the process *nil* (see Figures 6.9A and 6.34A). The rule adds a self-edge Terminated for a BB_CCS-node (the edge did not exists before), and creates a Nil-node, with the label Process that means that the Nil-node inherits a Process-node. Then the tggEmpty rule requires also the existence of a corresponding BB_PN-node (there exists a CN-node which connects BB_CCS- and BB_PN-nodes). The rule creates a self edge Terminated for a BB_PN-node.

Table 6.3: Transformation rules and their tasks

Rule name	**Task**
tggInitial	Creates a skeleton, i.e. the first BBs: a BB_CCS-node and a BB_PN-node.
tggEmpty	Turns a CCS BB into a structure, which corresponds to CCS process *nil*, and a Petri net BB into a structure, which consists of a single Place-node (A $\leftrightarrow$ A).
tggSummation	Creates a CCS graph, which corresponds to pattern B for a well-formed CCS process (P_1+P_2), and pattern B for a Petri net graph, where the token could flow through two different paths (B $\leftrightarrow$ B).
tggComposition	Creates a CCS graph, which corresponds to pattern C for a well-formed CCS process ($P_1 \mid P_2$), and pattern C for a Petri net graph (C $\leftrightarrow$ C).
tggSequence	Creates a CCS graph, which corresponds to pattern D for a well-formed CCS process ($a.P_1$), and pattern D for a Petri net graph, where a transition follows a BB (D $\leftrightarrow$ D).
tggSequenceCoAction	Creates a co-event in CCS graph and a corresponding transition in Petri net with an extra transition, which corresponds to τ.
tggVariableDeclaration	Creates a CCS graph, which corresponds to CCS process $\mu x.P_1$ (E $\leftrightarrow$ $\emptyset$ (but due to corresponding nodes the spot for recursion is traced)).
tggRecursion	Creates a CCS graph, which corresponds to CCS process x, and a recursive Petri net, by creating an edge to a certain Place-node (F $\leftrightarrow$ Definition 28).
tggRecursionI	Creates a CCS graph, which corresponds to CCS process x, and a recursive Petri net, by creating edges to Place-nodes marked as initial (F $\leftrightarrow$ Definition 29).

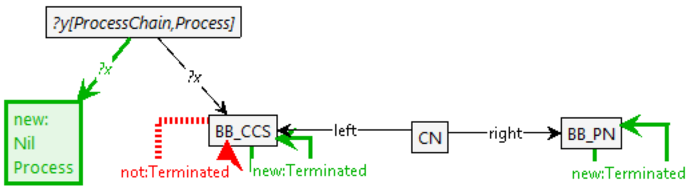


Figure 6.38: The tggEmpty rule (turns a CCS BB into a structure, which corresponds to the CCS process *nil*, and a Petri net BB into a structure, which consists of a single Place-node)

Important note

Note that later the node label $\{?y[ProcessChain, Process]\}$ means that the node could be either a ProcessChain- or a Process-node.

The edge label $?x$ means that whenever a label for the incoming edge to the BB_CCS-node is, the label for the edge marked as 'new' and $?x$ must be the same.

The tggSummation rule

The tggSummation rule builds a structure of the CCS well-formed graph, which corresponds to the CCS process $P_1 + P_2$. For this, the rule needs a match, which is a corresponding BB_CCS-node and a BB_PN-node without any Terminated-label. The rules creates a Summation-node, which is also labelled as Process, with two child BB_CCS-nodes. Then, in the target model, the rule creates two BB_PN-nodes, which are corresponding to the newly created BB_CCS-nodes. The input Place-node for the BB_PN-node in a match, is also the input Place-node for the newly created BB_PN-nodes.

The tggComposition rule

The tggComposition rule turns two corresponding BBs into a structure, which corresponds to the process $P_1 \mid P_2$. For this purpose, the rule requires a match that consists of two corresponding BB_CCS- and BB_PN-nodes, which have no Terminated-labels. The rule adds these labels to these BB_CCS- and BB_PN-nodes. In a CCS graph it also creates a Composition-node with two following BB_CCS-nodes. The Composition-node has the incoming edge from the same node as the BB_CCS-node in a match. This allows to keep a tree structure.

The construction in a Petri net graph is the following. The rule creates a structure shown in Figure 6.34C, which is two separate well-formed patterns. If the BB_PN-node in a match has either preceding or following Transition-nodes or Place-nodes marked as initial, then the newly created BB_PN-nodes have these preceding and following nodes too.

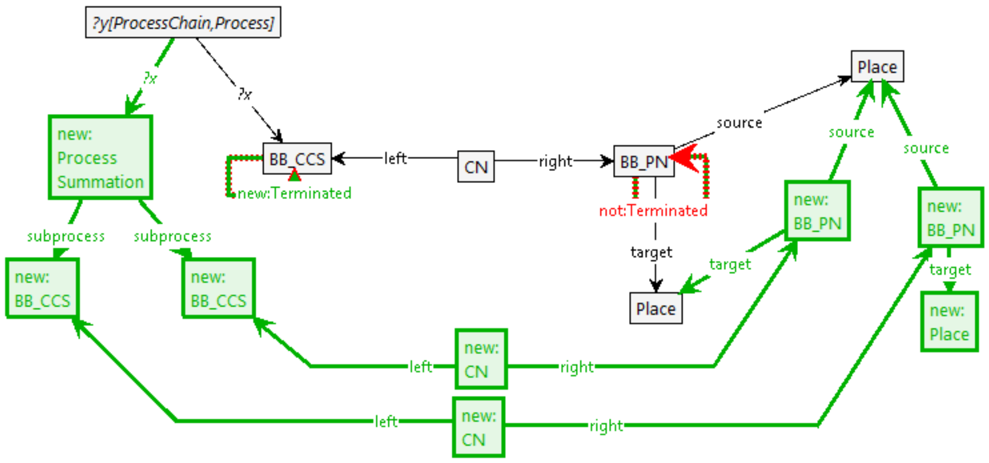


Figure 6.39: The tggSummation rule (creates a CCS graph, which corresponds to the CCS process $P_1 + P_2$, and a Petri net, where a token could flow through two different paths)

The tggSequence rule

The tggSequence rule converts a CCS BB into a graph, which corresponds to the process $a.P_1$, and a PN BB into a transition with a following BB, i.e. the structure from Figure 6.34D. For this purpose, the rule creates a self Terminated-edges for the corresponding BB_CCS- and BB_PN-nodes (the edges did not exist before). In addition, the rule creates a Sequence-node, with the Process-label, which means that the Sequence-node inherits a Process-node. Then the rule creates two child nodes for the Sequence-node: an Event-node and a BB_CCS-node. The Event-node has an attribute *name*, which value is calculated during the rule application. Here, the attribute type is not a string, but an integer. This allows to create a unique name for each event. The rule says that there must not exist a node with an attribute *name*, which value is bigger then a value of some one particular node. Therefore, the attribute with the highest value is found. The tggSequence rule adds the integer “1" to the highest existing attribute value in a graph – that is how the newly created Event-node gets its unique attribute value.

In addition, the tggSequence rule creates a Transition-node, which has a common input Place-node in with the BB_PN-node in the match. The Transition-node is a corresponding node for the newly created Event-node. Therefore, there exists a CN-node, which connects them. The rule creates a chain of nodes: a Place-node, and another BB_PN-node. The chain of nodes is closed with an existing Place-node, which is an outgoing for the existing BB_PN-node. The two newly created BB_CCS- and BB_CCS-nodes are connected with a CN-node.

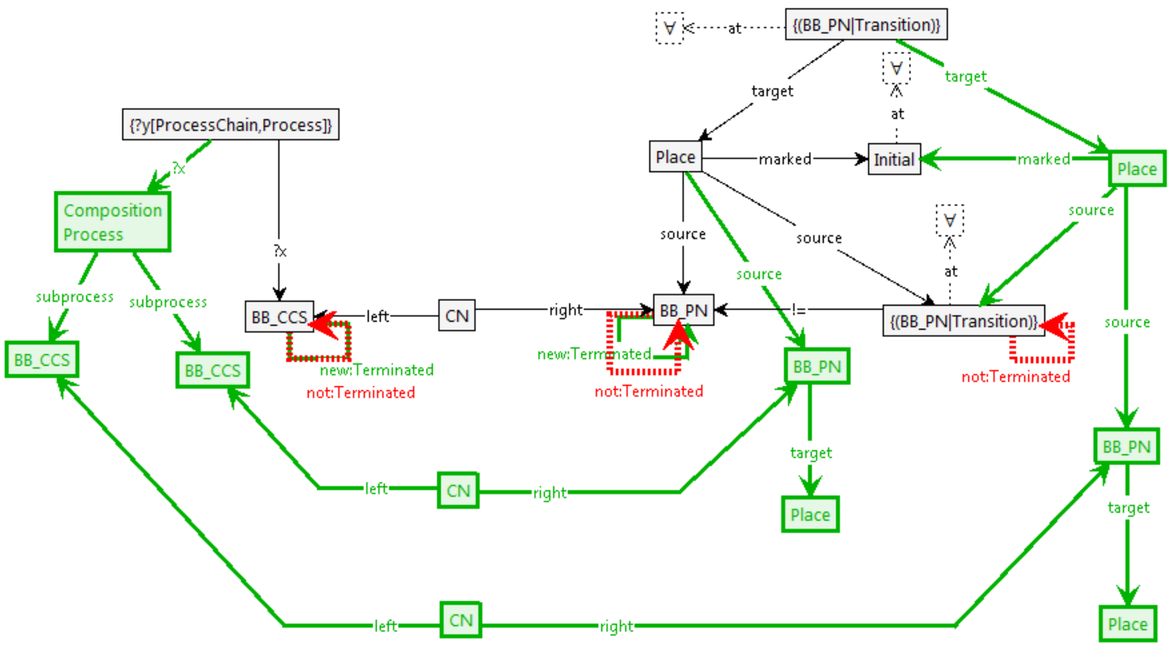


Figure 6.40: The tggComposition rule (creates a CCS graph, which corresponds to pattern C for the well-formed CCS process $P_1 \mid P_2$, and the pattern C for a Petri net graph)

The tggSequenceCoAction rule

The tggSequenceCoAction rule builds a structure of the CCS well-formed graph, which corresponds to the CCS process $a.P_1 \,|\, \bar{a}.P_2$, and a Petri net structure, which is depicted in Figure 6.35. For this, the rule requires a match, which is a corresponding BB_CCS-node and a BB_PN-node without a Terminated-label, beside this, the match consists of an Event-node and a Transition-node, which are connected by a CN-node. The rule creates a Sequence-node (which is also labelled as Process) with a child Event-node, which has an attribute *co-action* with a value of the existing node *name* attribute. The Sequence-node has a second child node of the type BB_CCS and is connected with a predecessor of the BB_CCS-node in the match that allows to keep a tree structure of a tree.

The rule also creates two Transition-nodes: one is the corresponding for the Event-node (i.e. the event $\bar{a}$) and is followed by a PN BB, another Event-node corresponds to a tau-event (it has a special meaning that the events a and $\bar{a}$ happen simultaneously). The former Transition-node has the same attribute value as the corresponding Event-node, but with the prefix “co”. The later Transition-node shares input and outgoing Place-nodes with the other two Transition-nodes (one the former and another form the match) and has an attribute with the name ‘tau‘. The quan-

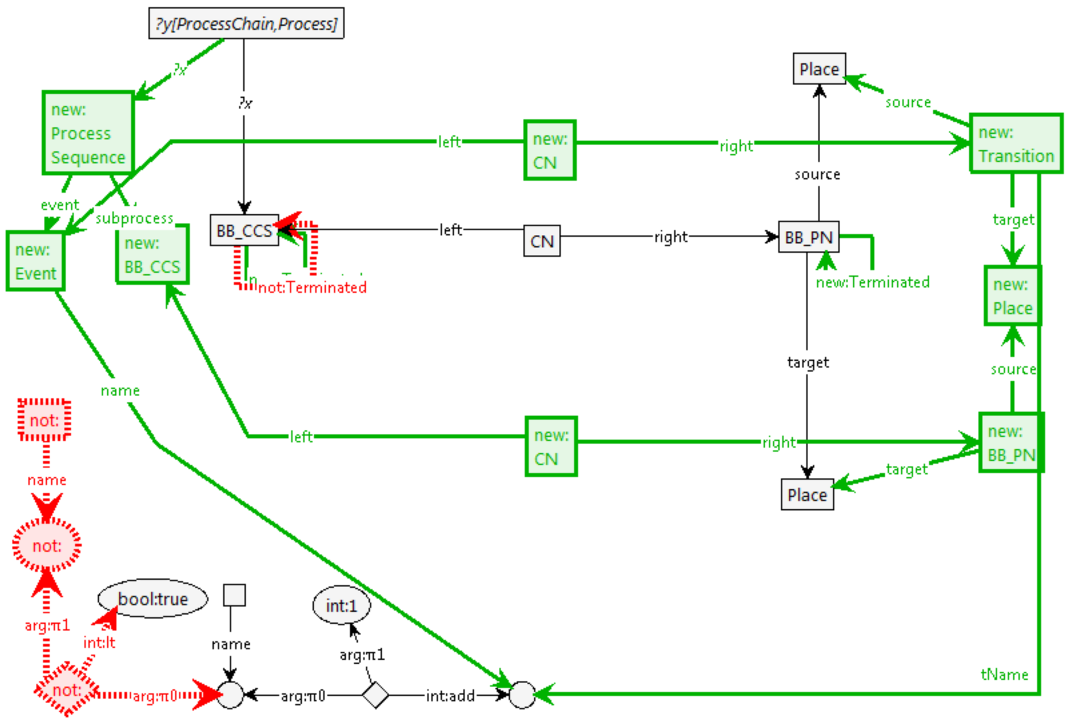


Figure 6.41: The tggSequence rule (creates a CCS graph, which corresponds to the CCS process $a.P_1$, and a Petri net transition, which follows a BB)

tifies could be read as follows: for all Place-nodes followed by the Transition-node in the match, the newly created Transition-node, which corresponds to a tau-event, is also connected with these Place-nodes.

The tggVariableDeclaration rule

The tggVariableDeclaration rule turns a CCS BB into a structure which corresponds to the process $\mu x.P_1$ and does not change the Petri net graph (it is supposed that a correspondence CN-node keeps a track of the future start for a recursive loop). As usually, the rule requires a match with corresponding BB_CCS- and BB_PN-nodes without Terminated-labels. The BB_CCS-node gets a Terminated-label, instead a tree structure with a root node, which precedes the BB_CCS-node, is created. This structure consists of a Variable-node with an unique attribute value. The Variable-node has a child BB_CCS-node, which plays further a role of a corresponding node for the BB_PN-node. Another child node of the Variable-node is a Mu-node, which denotes a beginning of the recursive loop.

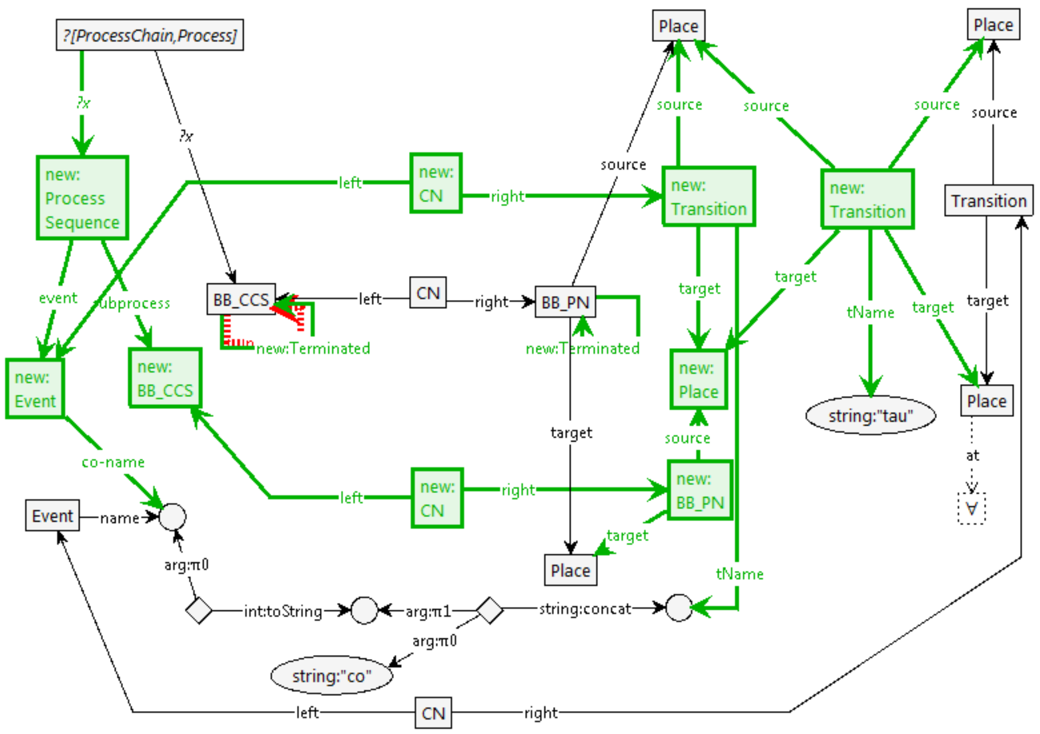


Figure 6.42: The tggSequenceCoAction rule (creates a co-event in a CCS graph and a corresponding transition in a Petri net with an extra transition, which corresponds to τ, which means that an event and a co-event are executed simultaneously)

The tggRecursion rule

The tggRecursion rule creates a recursive structure, which was defined in Definition 28. For this, the rule requires a match of two pairs of corresponding BB_CCS- and BB_PN-nodes: first pair with Terminated-label, the second – without. The BB_CCS-node from the first pair must be connected with a Recursion-node, which is by turn connected with a Variable-node. The rule creates another Variable-node, which has the same attribute value as the existing one, and it is connected with the same predecessor as the BB_CCS-node from the second pair. A recursive loop in Petri nets is implemented by creation of the target-edge from all Transition- and BB_PN-nodes, which precedes the BB_PN-node from the second pair, to the Place-nodes, which precedes the BB_PN-node from the first pair.

The tggRecursionI rule

The tggRecursionI rule creates a recursive structure, which was defined in Definition 29. For this, the rule requires a match of a pair of corresponding BB_CCS-

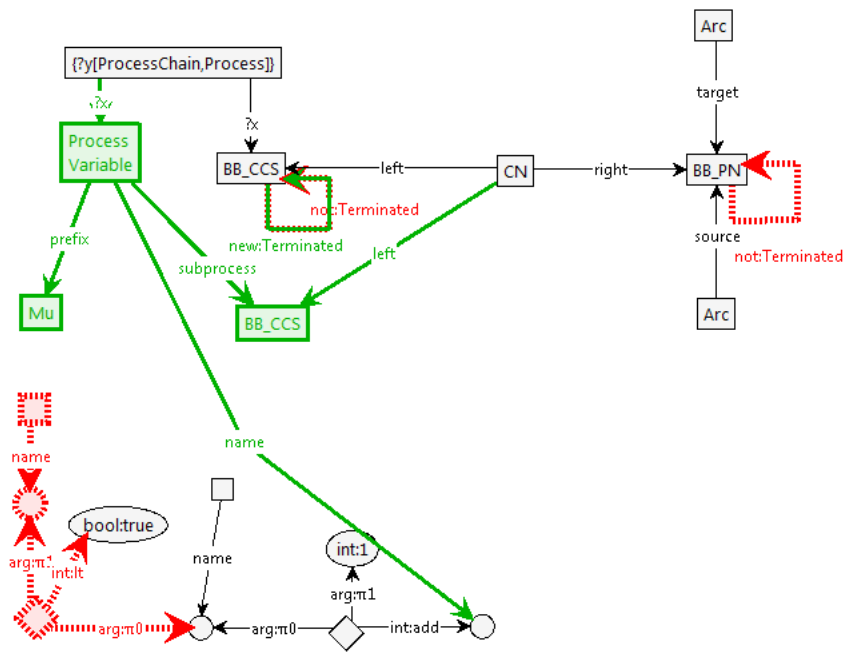


Figure 6.43: The tggVariableDeclaration rule (creates a CCS graph, which corresponds to the CCS process $\mu x.P_1$)

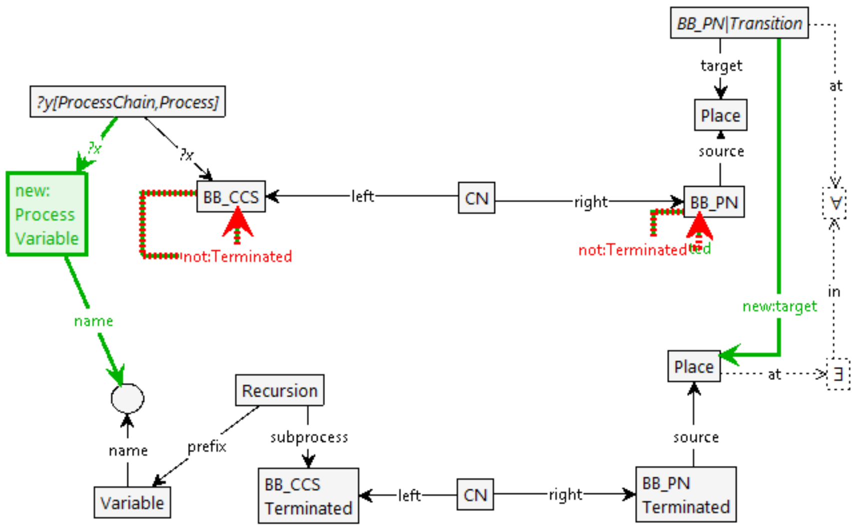


Figure 6.44: The tggRecursion rule (creates a CCS graph, which corresponds to CCS process x, and a recursive Petri net from Definition 28)

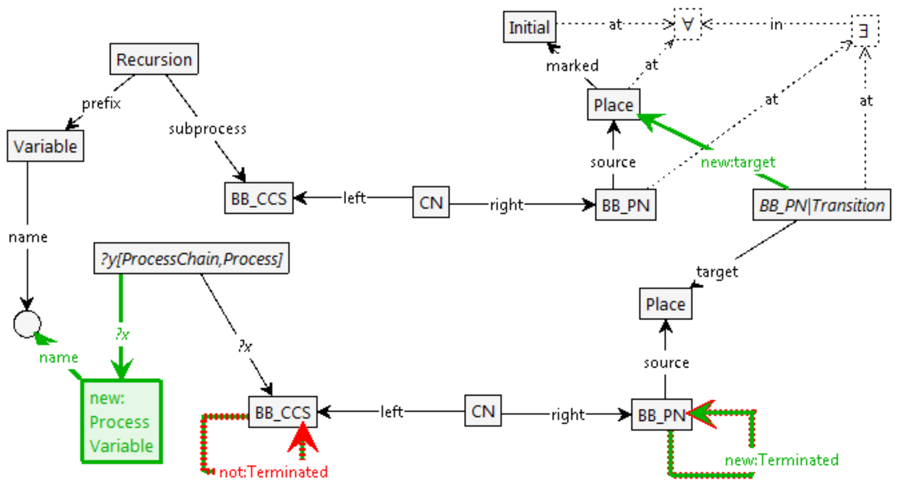


Figure 6.45: The tggRecursionI rule (creates a CCS graph, which corresponds to CCS process x, and a recursive Petri net from Definition 29)

and BB__PN-nodes without a Terminated-label. Additionally, the rule requires the existence of a Recursion-node with a child Variable-node and a child BB__CCS-node. The later must be a corresponding node for some BB__PN-node, which has an input Place-node marked as initial. The rule creates a Variable-node with the same attribute value as the Variable-node in the match. Then the rule creates a target-edge between the BB__PN- or Transition-node, which precedes the BB__PN-node in the match, and all Place-nodes marked as initial.

6.4.4 Auxiliary Notation for CCS Graphs

We introduce some notation for the structure of CCS graphs, which are built by the TGG rules from MT_{CCS2PN}. Since the CCS graph has always a tree structure (the fact is easily proven by the induction of the TGG rules), where a Process-node v_{Proc} is a root element and each node except the root node has a Process-node as a parent node, we can define the conditions on the paths of Process-nodes in a CCS graph. The idea is to separate paths in a such way, that each path either encloses only one Sequence-node, or encloses a leaf node of a tree. Recall that a Sequence-node is also a Process-node.

We specify three types of paths. The first condition condA is for the paths, where the first node in a path is either a root of the whole graph or a follower of a Sequence-node, the last node is of a Sequence type and all the others have the types different to Sequence. The only exception is when the first node of a path is a Sequence-node, then the path consists of one node. The second condition condB is for the paths, where the first node is connected to a Sequence-node (which is a predecessor), the last node is a leaf of a tree and a Nil-node. Finally, the third condition condC is for the paths, where the last node is a leaf, there is no node in the path is of a Sequence type, the first node is either a root of a tree or there is a Recursion-node in a path, which declares a variable with the same attribute as a leaf Variable-node.

(condA) We say that condA holds for the sequence of Process-nodes $\{v_i\}_{i=1}^N = \{v_1, \ldots, v_N\}$, denoted as condA($\{v_i\}_{i=1}^N$), if for all i:

$$subprocess(v_i, v_{i+1}) \bigwedge$$

$$v_N \text{ is a Sequence-node} \bigwedge$$

$$((\exists \text{ Sequence-node } v_0 \text{ such that } subprocess(v_0, v_1)) \vee root(v_1)) \bigwedge$$

$$\forall v_i \in \{v_2, \ldots, v_{N-1}\}\ v_i \text{ is not of the Sequence type} \bigwedge$$

$$(\text{if } v_1 \text{ is a Sequence-node then } N = 1)$$

(condB) We say that condB holds for the sequence of Process-nodes $\{v_i\}_{i=1}^N = \{v_1, \ldots, v_N\}$, denoted as condB($\{v_i\}_{i=1}^N$), if for all i:

$$subprocess(v_i, v_{i+1}) \bigwedge$$

$$\exists \text{ a Sequence-node } v_0 \text{ such that } subprocess(v_0, v_1) \bigwedge$$

$$leaf(v_N = v_{Nil}) \bigwedge$$

$$\forall v_i \in \{v_1, \ldots, v_N\}\ v_i \text{ is not of the Sequence type}$$

(condC) We say that condC holds for the sequence of Process-nodes $\{v_i\}_{i=1}^N = \{v_1, \ldots, v_N\}$, denoted as condC($\{v_i\}_{i=1}^N$), if for all i:

$$subprocess(v_i, v_{i+1}) \bigwedge$$

$$\forall v_i \in \{v_1, \ldots, v_N\}\ v_i \text{ is not of the Sequence type} \bigwedge$$

$$(root(v_1) \vee$$

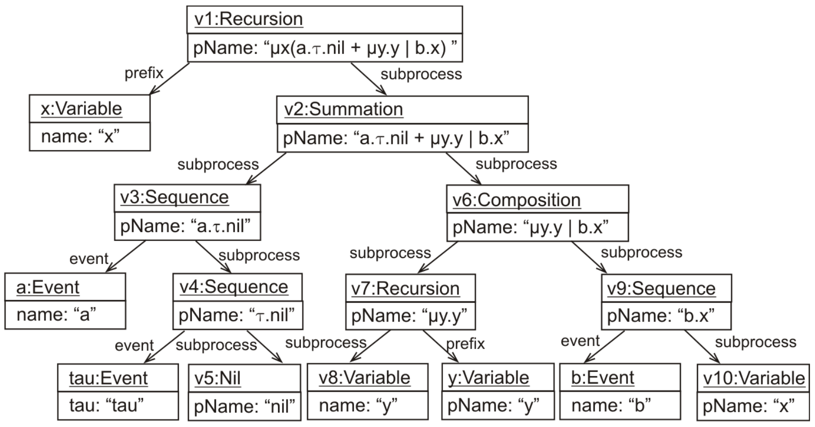


Figure 6.46: The CCS graph for the process $P = \mu x(a.\tau.nil + \mu y.y|b.x)$

$$(\exists j, v_{Rec}, v'_{Var} : v_j = v_{Rec} \wedge prefix(v_{Rec}, v'_{Var}) \wedge pName(v'_{Var}) = name(v_N))) \bigwedge$$

$$leaf(v_N)$$

Remark. We cosider the paths $\{v_i\}_{i=1}^{N}$, where

1. $v_{Var} = v_N$,
2. $\exists v_{Seq} : subprocess(v_{Seq}, v_1)$,
3. $\forall v_i \in \{v_1, \ldots, v_N\}$, v_i is not of a Sequence type,
4. $\nexists j, v_{Rec} : v_j = v_{Rec} \wedge prefix(v_{Rec}, v'_{Var}) \wedge pName(v_{Var}) = name(v_N)$,

then the path $\{v_i\}_{i=1}^{N}$ satisfies the same conditions as a path $\{\tilde{v}_i\}_{i=1}^{N}$, where $\exists j, \tilde{v}_{Rec}$: $\tilde{v}_j = \tilde{v}_{Rec} \wedge prefix(\tilde{v}_{Rec}, v'_{Var}) \wedge (name(v'_{Var}) = pName(v_N))$.

Note that we consider paths with no repeated nodes.

We illustrate the defined conditions on a sample of a CCS graph. Figure 6.46 illustrates the CCS graph for the process $P = \mu x(a.\tau.nil + \mu y.y|b.x)$. Here, the condition condA holds for the paths $\{v_1, v_2, v_3\}$, $\{v_4\}$, $\{v_1, v_2, v_6, v_9\}$. The path $\{v_5\}$ satisfies the condition condB. There is the path $\{v_1, v_2, v_6, v_7, v_8\}$ for which the condition condC holds. Further, the paths $\{v_{10}, v_1, v_2, v_3\}$ and $\{v_{10}, v_1, v_2, v_6, v_9\}$ satisfy condA. Finally, the path $\{v_{10}, v_1, v_2, v_6, v_7, v_8\}$ satisfies the condition condC.

The next proposition claims that every Process-node in *any* CCS graph, is a part of a path, which satisfies one of the three conditions, i.e. either condA, condB, or condC.

Proposition 6.4.1. *Let $G_{CCS} \in \mathcal{G}^{\mathrm{rt}}_{CCS}$, v_{Proc} be a* Process*-node in G_{CCS}. Then there exists at least one path $\{v_i\}_{i=1}^{N}$ and a number $j : 1 \leq j \leq N$ such that $v_j = v_{Proc}$ and one of the conditions holds for this path:*

(A) $\mathsf{condA}(\{v_i\}_{i=1}^{N})$,
(B) $\mathsf{condB}(\{v_i\}_{i=1}^{N})$,
(C) $\mathsf{condC}(\{v_i\}_{i=1}^{N})$.

Proof. Let us consider only Process-nodes in a general structure of a CCS graph. Due to the TGG rules MT_{CCS2PN}, each Process-node has either one or two child Process-nodes, except leaves of the tree, and one predecessor Process-node. We analyse a random Process-node v_{Proc} and show that according to the TGG rules MT_{CCS2PN} the proof statement holds for this node.

The diagram below shows an example of a tree structure of Process-nodes.

$$\begin{tikzcd}
& v_{Proc} \arrow[d] & & & \\
& v_{Proc} \arrow[dl] \arrow[dr] & & & \\
v_{Proc} & & v_{Proc} \arrow[dl] \arrow[dr] & & \\
& \dots \arrow[dl] \arrow[dr] & & v_{Proc} \arrow[dr] & \\
v_{Proc} & & v_{Proc} \arrow[dr] & & v_{Proc} \\
& & & v_{Proc} &
\end{tikzcd}$$

Note that any v_{Proc} is either v_{Seq}, v_{Sum}, v_{Com}, v_{Rec}, v_{Nil}, or v_{Var}. Further, we consider that v^0_{Proc} is v_{Proc}, denoted later as $v^0_{Proc} := v_{Proc}$. Depending on the type of v_{Proc} as it is mentioned earlier, we examine a follower or predecessor of v_{Proc} (the examined node is denoted as v_{Proc}). We use also boolean variables $flagV$ and $flagV$ with initial values $flagV := false$ and $flagR := false$ for the case with a recursion.

We analyse each case separately.

1. $v^0_{Proc} = v_{Seq}$, then the Process-node belongs to the path condA,
2. $v^0_{Proc} = v_{Sum}$ or v_{Com}, then we analyse a follower of v^0_{Proc} denoted as v_{Proc},
3. $v^0_{Proc} = v_{Rec}$, then we analyse a follower of v^0_{Proc} denoted as v_{Proc} and $flagR := true$,
4. $v^0_{Proc} = v_{Var}$, then we analyse a predecessor of v^0_{Proc} denoted as v_{Proc} and $flagV := true$,
5. $v^0_{Proc} = v_{Nil}$, then we analyse a predecessor of v^0_{Proc} denoted as v_{Proc}.

Note that if v_{Proc} is a follower of v^0_{Proc} then $subprocess(v^0_{Proc}, v_{Proc})$, if v_{Proc} is a predecessor of v^0_{Proc} then $subprocess(v_{Proc}, v^0_{Proc})$.

Analysis of a follower:

1. $v_{Proc} = v_{Seq}$, then v^0_{Proc} is from the path condA.
2. $v_{Proc} = v_{Sum}$, then analyse a follower of v_{Sum}.
3. $v_{Proc} = v_{Com}$, then analyse a follower of v_{Com}.
4. $v_{Proc} = v_{Rec}$, then analyse a follower of v_{Rec} and $flagR := true$.
5. $v_{Proc} = v_{Var}$, if $flagR = true$ (the corresponding v_{Rec} was analysed), then the path belongs to condC, otherwise analyse a predecessor of v^0_{Proc} and $flagV := true$.
6. $v_{Proc} = v_{Nil}$, then analyse a predecessor of v^0_{Proc}.

Analysis of a predecessor:

1. $v_{Proc} = v_{Seq}$, if $flagV = true$, then analyse a follower of corresponding Recursion-node v_{Rec}, else v^0_{Proc} is from the path condB.
2. $v_{Proc} = v_{Sum}$, if v_{Sum} is a root of a tree, then v^0_{Proc} is from the path condC, otherwise we analyse a predecessor of v_{Proc}.
3. v_{Com}, if v_{Com} is a root of a tree, then v^0_{Proc} is from the path condC, otherwise we analyse a predecessor of v_{Proc}.
4. v_{Rec}, if v_{Rec} is a root of a tree or $flagV = true$, then v^0_{Proc} is from the path condC, otherwise we analyse a predecessor of v_{Rec}.
5. v_{Var} is impossible,
6. v_{Nil} is impossible.

The algorithm always terminates, because the only possible loop is a path with v_{Var}, which has a marker in case we meet the node a second time.

Remark. We consider only one follower, although there could be two (in cases v_{Sum} and v_{Comp}), it means that we choose a particular path, the second path could satisfy another condition. □

6.4.5 Important Observations about CCS Graphs

For proving the correctness of the model transformation MT_{CCS2PN} we need some observations about the structure of CCS graphs which are built by the TGG rules from MT_{CCS2PN}. The following propositions are based on simple observations about the way the TGG rules are applied.

The following proposition tells that that each Event-node in a CCS graph has an adjacent Process-node, which has an adjacent Process-node too.

Proposition 6.4.2. *Let $G_{CCS} \in \mathcal{G}^{\mathsf{st}}_{CCS}$, v_{Event} be an* Event*-node in G_{CCS}. Then $\exists! v_{Proc}$ and v'_{Proc} such that $event(v_{Proc}, v_{Event})$ and $subprocess(v_{Proc}, v'_{Proc})$.*

Proof. The tggSequence and tggSequenceCoAction rules are the only rules that create a Sequence-node in a CCS graph. This node is connected with a Process-node, which has by turn a child BB_CCS-node. The other rules from MT_{CCS2PN} do not change this structure, instead build another child node of the Process-type for the Sequence-node. □

The next two propositions state that the nodes in a path of Process-nodes, which satisfies the conditions condA, condB, condC, are of certain type.

Proposition 6.4.3. *Let* $\{v_i\}_{i=1}^{N}$ *be a path of* Process-*nodes such that* condA$(\{v_i\}_{i=1}^{N})$*, then* $v_i \in \{v_1, \ldots, v_{N-1}\}$ *is either a* Summation-*,* Composition - *,* Recursion- *or* Variable-*node, and* v_N *is a* Sequence-*node.*

Proposition 6.4.4. *Let* $\{v_i\}_{i=1}^{N}$ *be a path of* Process-*nodes such that* condB$(\{v_i\}_{i=1}^{N})$ *or* condC$(\{v_i\}_{i=1}^{N})$*, then* $v_i \in \{v_1, \ldots, v_{N-1}\}$ *is either a* Summation-*node, or a* Composition-*node, or a* Recursion-*node, and* v_N *is a* Nil-*node.*

Proof. The rules from MT_{CCS2PN} build a tree structure from Process-nodes. It can be easily proven by considering every rule from MT_{CCS2PN}. Then Propositions and follows by the definition of conditions condA, condB and condC. □

The next proposition is a single observation about adjacent nodes of Summation-, Composition-, Recursion- and Variable-nodes.

Proposition 6.4.5. *Let* $G_{CCS} \in \mathcal{G}_{CCS}^{\mathsf{st}}$*,*

(A) let v *be either a* Summation-*node or a* Composition-*node in* G_{CCS}*, then* $\exists! v_{Proc}^{1}$ *and* v_{Proc}^{2} : $subprocess(v, v_{Proc}^{1}) \wedge subprocess(v, v_{Proc}^{2})$.

(B) let v *be a* Recursion-*node in* G_{CCS}*, then* $\exists! v_{Proc} : subprocess(v, v_{Proc})$.

(C) let v *be a* Variable-*node in* G_{CCS}*, then* $\exists! v_{Proc}, v_{Rec}$: $subprocess(v_{Rec}, v_{Proc}) \wedge prefix(v_{Rec}, v'_{Var}) \wedge pName(v) = name(v'_{Var})$.

Proof. The proof is a consequence of the tggSummation, tggComposition and tggVariableDeclaration rules and the fact that the other rules do not change this structure. □

The following observation concerns a run-time graph, it says that during the semantics rules are applied a Current-node is always connected with a Mark-node.

Proposition 6.4.6. *Let* $G_{CCS} \in \mathcal{G}_{CCS}^{\mathsf{rt}}$*,* v *is a* Current-*node in* G_{CCS}*, then* $\exists v_{Mark} : mark(v)$.

6.5 Correctness of Model Transformation (Step 5)

In this section we want to show that the model transformation MT_{CCS2PN} is behaviour preserving. Recall that semantics preserving in the context of this thesis means the behaviour preservation. Let $Q(G_{CCS})$ be a transition system which is a result of application of semantic rules from $\mathcal{RS}_{CCS}$ to the graph $G_{CCS} \in \mathcal{G}^{\mathsf{st}}_{CCS}$. Let $G_{PN} \in \mathcal{G}^{\mathsf{st}}_{PN}$ be a Petri net graph such as $MT_{CCS2PN}(G_{CCS}, G_{PN})$ and $Q(G_{PN})$ be a transition system generated for the graph G_{PN} by applying the semantic rules from $\mathcal{RS}_{PN}$. The semantics preservation means that for every behavioural property φ in the CCS language and its interpretation χ for the Petri net language. The following statement holds:

$$Q(G_{CCS}) \models \varphi \quad \Rightarrow \quad Q(G_{PN}) \models \chi(\varphi) \tag{6.2}$$

We show a statement, which implies 6.2, i.e. that $Q(G_{CCS})$ and $Q(G_{PN})$ are weakly bisimilar in respect to the mappings map_{CCS} and map_{PN}:

$$map_{CCS}(Q(G_{CCS})) \approx map_{PN}(Q(G_{PN})) \tag{6.3}$$

here the mappings map_{CCS} and map_{PN} are defined in Section 6.3. However, we still need to define the relation $\mathcal{R}_{CCS2PN}$ (i.e. the relation $\approx$). We start by introducing some notation and important observations, which concern correspondences created during model transformations MT_{CCS2PN}. The observations will help us to specify the relation $\mathcal{R}_{CCS2PN}$.

6.5.1 Auxiliary Notation for Corresponding Nodes

We continue with notation for corresponding nodes in a TGG graph G_{CCS2PN}. For Event-node v_{Event} and a Transition-node v_{Trans}, we write $cn(v_{Event}, v_{Trans})$ if there is a CN-node v_{cn} and a left-edge from v_{CN} to v_{Event} and a right-edge from v_{CN} to v_{Trans}.

The TGG rules generate correspondences between Event- and Transition-nodes, i.e. $cn(v_{Event}, v_{Trans})$. Further, we define the notation for Process- and Place-nodes which could be related due the connection with corresponding Event-nodes and Transition-nodes.

(cn1) We write $cn1(v_{Proc}, v_{Place})$ for Process- and Place-nodes, if there exists an Event-node v_{Event} such that $event(v_{Proc}, v_{Event})$, and there exists a Transition-node v_{Trans} such that $source(v_{Trans}, v_{Place})$, and $cn(v_{Event}, v_{Trans})$.

(cn2) We write $cn2(v_{Proc}, v_{Place})$ for Process- and Place-nodes, if there exist a Process-node v'_{Proc} and an Event-node v_{Event} such that $event(v'_{Proc}, v_{Event}) \wedge subrocess(v'_{Proc}, v_{Proc})$, and there exists a Transition-node v_{Trans} such that $target(v_{Trans}, v_{Place})$, and $cn(v_{Event}, v_{Trans})$.

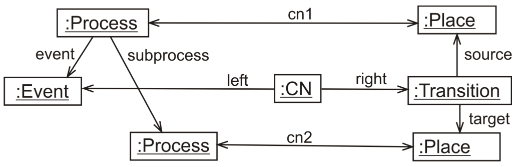


Figure 6.47: Relevant correspondences: cn, $cn1$ and $cn2$

Figure 6.47 illustrates the relevant correspondences that are crucial parts of our proof. The correspondence cn is defined over Event-nodes and Transition-nodes by a CN-node, which is built by the TGG rules. The correspondences $cn1$ and $cn2$ are defined over the Process- and Place-nodes.

6.5.2 Important Observations about Corresponding Structure

For the proof we firstly provide some observations about the correspondences generated by the TGG rules, despite the fact that the semantics rules are applied on the individual models. In addition, we use the observation that the syntactic structure is kept when the semantics rules are applied to a CCS graph.

First observation Both for well-formed CCS and Petri nets graphs, the semantics rules keep the syntactic structure of a graph, i.e. all Process- and Event-nodes, all Transition-, Place- and Initial-nodes stay the same.

We formalize the first observation that the syntactic structure of graphs stays the same when semantic rules are applied.

Proposition 6.5.1. *Let $G_{CCS} \in \mathcal{G}^{\mathsf{rt}}_{CCS}$ be a CCS graph. If $G_{CCS} \xrightarrow{r} G'_{CCS}$ for some $r \in \mathcal{RS}_{CCS}$ then $\pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{CCS}}\backslash\mathsf{dyn}}(G_{CCS}) = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{CCS}}\backslash\mathsf{dyn}}(G'_{CCS})$, where $\mathsf{dyn} = \{\mathsf{Current}, \mathsf{Mark}\}$ is a set of dynamic elements, i.e.* Current- *and* Mark-*nodes, and* T\dyn *is the type* T *without the* dyn-*elements.*
A corresponding property holds for a Petri net graph.

The next observation shows that correspondences between Event and Transition are unique.

Proposition 6.5.2. *Let $G \in \mathcal{G}_{CCS2PN}$, v_{Event} an* Event-*node and v_{Trans} a* Transition-*node in G. Then the following two properties hold:*

(A) $\exists! v$ of the Event *type such that $cn(v, v_{Trans})$ and $tName(v_{Trans}) = name(v)$ (the same attribute values),*

(B) $\exists! v$ of the Transition *type such that $cn(v_{Event}, v)$ and $tName(v) = name(v_{Event})$ (the same attribute values).*

Proof. The tggSequence and tggSequenceCoAction rules are the only rules in MT_{CCS2PN}, which create the required structure in G. Since the other rules from MT_{CCS2PN} do not affect it, the proposition is proven. $\square$

Second observation Correspondences between nodes in CCS models and Petri nets models are kept during application of the semantic rules. Predicates *cn*, *cn*1, *cn*2 as well as Proposition 6.5.2 can thus also be applied to separate CCS and Petri nets graphs.

Further propositions are about the structure of corresponding nodes and the adjacent nodes. The following proposition illustrates the existence of correspondences for each Place-node.

Proposition 6.5.3. *Let* $G_{PN} = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{PN}}}(G_{PN2CCS})$ $G_{CCS} = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{CCS}}}(G_{PN2CCS})$, v_{Place} *be a* Place*-node in* G_{PN}*. Then one of the following properties hold:*

(A) $\exists!$ Process*-node* $v \in G_{CCS}$ *such that* $cn1(v, v_{Place})$*,*

(B) $\exists!$ Process*-node* $v \in G_{CCS}$ *such that* $cn2(v, v_{Place})$ *and* $\neg\exists\ v_{Proc} : cn1(v_{Proc}, v_{Place})$*,*

(C) $\exists!$ Process*-node* $v \in G_{CCS}$ *such that* $(v \in \{v_i\}_{i=1}^{N}) \wedge condC(\{v_i\}_{i=1}^{N}) \wedge (\neg\exists v_{Proc} : cn1(v, v_{Place}) \vee cn2(v, v_{Place}))$*.*

Proof. The proposition is easily proven since each Transition-node has a corresponding Event-node (Proposition 6.5.2). Then each Place is connected to a Transition-node, it means that there exists an edge either $source(v_{Place}, v_{Trans})$ or $target(v_{Place}, v_{Trans})$. To fulfil conditions *cn*1 and *cn*2 we need Proposition 6.4.2 which says that there are always two Process-nodes for an Event-node. If there are no Transition- and Event-nodes in the graphs G_{PN}, G_{CCS}, then there are paths in the G_{CCS}, where there are no Sequence-nodes and therefore the condition condC holds. □

We continue with observations concerning correspondences. We formalize the observation about the existence of correspondences for Process-nodes.

Proposition 6.5.4. *Let* $G_{PN}, G_{CCS} \subset \mathcal{G}^{\mathsf{rt}}_{PN2CCS}$*, then for each* Process*-node* v_{Proc} *in* G_{CCS} *one of the following properties hold:*

(A) if v_{Proc} *belongs to the path* $\{v_i\}_{i=1}^{N}$ *such that* $\exists j : v_{Proc} = v_j$ *and the condition* condA$(\{v_i\}_{i=1}^{N})$ *holds. Then, there exists a* Place*-node* $v_{Place} \in G_{PN}$ *such that* $cn1(v_N, v_{Place})$*,*

(B) if v_{Proc} *belongs to the path* $\{v_i\}_{i=1}^{N}$ *such that* $\exists j : v_{Proc} = v_j$ *and the condition* condB$(\{v_i\}_{i=1}^{N})$ *holds. Then, there exists a* Place*-node* $v_{Place} \in G_{PN}$ *such that* $cn2(v_1, v_{Place})$*,*

(C) if v_{Proc} *belongs to the path* $\{v_i\}_{i=1}^{N}$ *such that* $\exists j : v_{Proc} = v_j$ *and the condition* condC$(\{v_i\}_{i=1}^{N})$ *holds. Then, there exists a* Place*-node* $v_{Place} \in G_{PN}$ *such that either* $initial(v_{Place})$ *or* $cn2(v_1, v_{Place})$*.*

The following two propositions concern a Process-node, which is a root of a tree structured CCS graph, and a Place-node marked as initial.

Proposition 6.5.5. *Let $G_{CCS} \in \mathcal{G}_{CCS2PN}$, v_{Proc} a* Process*-node s.t $root(v_{Proc})$ in G. Then the one of the following two properties hold:*

(A) $\forall\{v_i\}_{i=1}^N$ – a sequence of Process*-nodes such that $(v_1 = v_{Proc}) \wedge$* condA$(\{v_i\}_{i=1}^N)$ *$\exists v$ of the* Place *type such that $cn1(v_N, v) \wedge initial(v_{Place})$,*

(B) $\forall\{v_i\}_{i=1}^N$ – a sequence of Process*-nodes such that $(v_1 = v_{Proc}) \wedge$* condC$(\{v_i\}_{i=1}^N)$ *$\exists v$ of the* Place *type such that $initial(v_{Place})$.*

There are no other Place*-nodes marked as initial except those, which are described in (A) and (B).*

Proposition 6.5.6. *Let $G_{PN} \in \mathcal{G}_{CCS2PN}$, v_{Place} a* Place*-node such that $initial(v_{Place})$ in G. Then the following two properties hold:*

(A)$\exists\{v_i\}_{i=1}^N$ – a sequence of Process*-nodes such that $(v_1 = v_{Proc})\wedge$* condA$(\{v_i\}_{i=1}^N)\wedge$ *$cn1(v_N, v_{Place})\wedge$ $root(v_1)$,*

(B)$\exists\{v_i\}_{i=1}^N$ – a sequence of Process*-nodes such that $(v_1 = v_{Proc}) \wedge$* condC$(\{v_i\}_{i=1}^N)\wedge$ *$root(v_1)$.*

The next proposition tells that if there exists a path between two Event-nodes then there exists a path between the corresponding Place-nodes. Note that the case (B) considers a recursion.

Proposition 6.5.7. *Let $G_{PN} \in \mathcal{G}_{CCS2PN}$, v_{Event}, v'_{Event} be* Event*-nodes in G_{CCS} and v_{Trans}, v'_{Trans} be* Transition*-nodes in G_{PN} such that $cn(v_{Event}, v_{Trans})$ and $cn(v'_{Event}, v'_{Trans})$.*

(A) If $\exists\{v_i\}_{i=1}^N$ – a path of Process*-nodes such that $event(v_N, v'_{Event})$ and there is a node v_{Seq} : $subprocess(v_{Seq}, v_1)\wedge$ $event(v_{Seq}, v_{Event})$ or*

(B) If there are two paths of Process*-nodes: (1) $\{v_i^1\}_{i=1}^N$ such that there is some node v_{Seq}^1 : $subprocess(v_{Seq}^1, v_1^1)\wedge$ $event(v_{Seq}^1, v_{Event})\wedge$ $v_N^1 = v_{Var}^1$, (2) $\{v_i^2\}_{i=1}^N$ such that $(v_{Seq}^2 = v_N^2)\wedge$ $event(v_{Seq}^2, v'_{Event})\wedge$* condA *and there exists a* Recursion*-node $v_{Rec} = v_j^2$ such that $prefix(v_{Rec}, v_{Var}^2)\wedge$ $name(v_{Var}^2) = pName(v_{Var}^1)$, then*
$\exists!$ v_{Place} – a Place*-node such that $target(v_{Trans}, v_{Place})$ and $source(v'_{Trans}, v_{Place})$.*

The following proposition is the reverse statement for Proposition 6.5.7.

Proposition 6.5.8. *Let $G_{PN} \in \mathcal{G}_{CCS2PN}$, v_{Event}, v'_{Event} in G_{CCS} and v_{Trans}, v'_{Trans} in G_{PN} such that $cn(v_{Event}, v_{Trans})$ and $cn(v'_{Event}, v'_{Trans})$. If $\exists v_{Place}$ such that $target(v_{Trans}, v_{Place})$ and $source(v'_{Trans}, v_{Place})$ then one of the following statements hold:*

(A) $\exists!$ path $\{v_i\}_{i=1}^N$ such that condA$(\{v_i\}_{i=1}^N) \wedge cn1(v_1, v_{Place}) \wedge cn2(v_N, v_{Place})$, *or*

(B) $\exists$ two paths: (1) $\{v_i^1\}_{i=1}^N$ such that $v_N = v_{Var}$ and $\exists v_{Seq}^1$: $subprocess(v_{Seq}^1, v_1^1)$, $event(v_{Seq}^1, v_{Event})$; (2) $\{v_i^2\}_{i=1}^N$ such that condA$(\{v_i^2\}_{i=1}^N) \wedge$ *$cn2(v_1, v_{Place}) \wedge v_N = v_{Var}$).*

6.5.3 Definition and Proving of Weak Bisimulation

We summarize the results of this section and previous observations in order to define the relation $\mathcal{R}_{CCS2PN}$, which consists of syntactic and semantic restrictions on the graphs G^{rt}_{CCS} and G^{rt}_{PN}.

Proposition 6.5.1 helps us to specify the condition (1) on a syntactic structure of the graphs. It says that the projection of graphs on the static type graphs is always the same during the application of the semantic rules.

To specify the conditions for run-time properties on corresponding nodes, we reason as follows. Since only Process-nodes could be connected with Current-nodes during the semantic rules application and due to Proposition 6.4.1 every Process-node belongs to one of the paths condA, condB, condC, then we can specify the restrictions on the run-time structure, by defining a location of a Current-node in a CCS graph. Due to the fact that only *places* can carry *tokens*, i.e. Place-nodes could be connected with Token-nodes, and Proposition 6.5.3 about the connection of Place-nodes and Process-nodes, we can specify three conditions on the run-time structure of a graph G^{rt}_{PN}. So we have the conditions (2)-(4) for run-time properties.

For the proof we construct the relation $\mathcal{R}_{CCS2PN}$ between the states of the CCS LTS and the Petri net LTS.

$$
\begin{aligned}
\mathcal{R}_{CCS2PN} \quad = \quad & (G_{CCS}, G_{PN}) \in \mathcal{G}^{\mathsf{rt}}_{CCS} \times \mathcal{G}^{\mathsf{rt}}_{PN} \mid \exists G_{CCS2PN} \\
& (1)(\pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{CCS}} \setminus \mathsf{dyn}}(G_{CCS}) = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{CCS}} \setminus \mathsf{dyn}}(G_{CCS2PN})) \wedge \\
& (\pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{PN}} \setminus \mathsf{dyn}}(G_{PN}) = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{PN}} \setminus \mathsf{dyn}}(G_{CCS2PN})), \\
& \text{where } \mathsf{dyn} \in \{v_{Curr}, v_{Mark}, v_{Tok}\} \text{ and } \mathsf{T} \setminus \mathsf{dyn} \text{ is } \mathsf{T} \text{ without } \mathsf{dyn}, \\
& (2)\ \forall v_{Place} \in G_{PN}, \{v_i\}_{i=1}^{N} \in G_{CCS} \text{ such that} \\
& \mathsf{condA}(\{v_i\}_{i=1}^{N}) \wedge cn1(v_N, v_{Place}) \text{ then} \\
& \exists j : current(v_j) \Leftrightarrow token(v_{Place}), \\
& (3)\ \forall v_{Place} \in G_{PN}, \{v_i\}_{i=1}^{N} \in G_{CCS} \text{ such that} \\
& \mathsf{condB}(\{v_i\}_{i=1}^{N}) \wedge cn2(v_1, v_{Place}) \text{ then} \\
& \exists j : current(v_j) \Leftrightarrow token(v_{Place}), \\
& (4)\ \forall v_{Place} \in G_{PN}, \{v_i\}_{i=1}^{N} \in G_{CCS} \text{ such that} \\
& (4a)\ initial(v_{Place}) \wedge root(v_1) \wedge \mathsf{condC}(\{v_i\}_{i=1}^{N}) \text{ or} \\
& (4b)\ \mathsf{condC}(\{v_i\}_{i=1}^{N}) \wedge cn2(v_1, v_{Place}) \text{ then} \\
& \exists j : current(v_j) \Leftrightarrow token(v_{Place}).
\end{aligned}
$$

The relation contains all pairs of CCS and Petri nets pairs which (1) in their syntactic structure follow the structure generated by the TGG rules, (2)-(4) exhibit run-time properties.

Theorem 6.5.9. Given MT_{CCS2PN} (as defined in Figures 6.37-6.45) and the relation $\mathcal{R}_{CCS2PN}$. Let G_{CCS} and G_{PN} be a CCS graph and a Petri net graph, respectively, such that $MT_{CCS2PN}(G_{CCS}, G_{PN})$. Then the relation $\mathcal{R}_{CCS2PN}$ $(\approx)$ is a weak bisimulation , i.e.

$$map_{CCS}(Q(G_{CCS})) \approx map_{PN}(Q(G_{PN}))$$

All presented propositions are essential for the showing that the relation $\mathcal{R}_{CCS2PN}$ given earlier in this section indeed defines a weak bisimulation.

Proof. of Theorem 6.5.9. Taking the relation $\mathcal{R}_{CCS2PN}$, we need to show the property of mutual simulation. We start with the requirement of initial states being in the relation. The initial states of the LTSs are G^0_{CCS} and G^0_{PN} and they satisfy the conditions of $\mathcal{R}_{CCS2PN}$ since they are directly generated by projection from the combined graph (condition (1)), since there are no run-time elements such as Current, Mark and Token, so conditions (2)-(5) are trivially satisfied.

Now assume $(G_{CCS}, G_{PN}) \in \mathcal{R}_{CCS2PN}$ and $G_{CCS} \xrightarrow{r_1} G'_{CCS}$. As we are looking at the LTSs with labels renamed according to map_{CCS} and map_{PN}, r_1 (the label of the transition) in principle is either Initial, Action or τ. We need to show that there is some G'_{PN} such that $G_{PN} \overset{\widehat{r_1}}{\Longrightarrow} G'_{PN}$ with $(G'_{CCS}, G'_{PN}) \in \mathcal{R}_{CCS2PN}$. However, as we are interested in the particular semantic rule applied during the step, we will instead directly look at the original LTSs and show that map_{CCS} and map_{PN} map rule names to the same label.

$r_1 =$ ccsInitial. Let $\langle L_1, R_1, \mathcal{N}_1\rangle$ be the ccsInitial rule (see Figure 6.16). If r_1 is applicable in G_{CCS}, we have a match $m_1 : L_1 \rightarrow G_{CCS}$, i.e., a node v_{Proc} such that $root(v_{Proc})$. Due to Proposition 6.4.1 there exists a path of Process-nodes $\{v_n\}_{n=1}^N$ such that $v_1 = v_{Proc}$ and either the condition condA or condC hold for $\{v_i\}_{i=1}^N$. Due to Proposition 6.5.5 for every root node there exists a Place-node marked as initial. Therefore in the graph G^{rt}_{PN} we have at least one Place-node $v_{Place,i}$, here $i \geq 1$.
From this, we construct a match $m_2 : L_2 \rightarrow G_{PN}$ for the rule $r_2 =$ pnInitial (both being mapped to Initial by map_{CCS} and map_{PN}) being defined as $\langle L_2, R_2, \mathcal{N}_2\rangle$. The match m_2 maps all Place-nodes $v_{Place,i}$ (for $i > 0$) marked as initial in L_2, i.e. $initial(v_{Place,i})$.
Thus, pnInitial is applicable in G_{PN}. Once the rules are applied, we have a graph G'_{CCS} with one Current-node such that $current(v_{Proc})$ and a graph G'_{PN} with Token-nodes such that $token(v_{Place,i})$ and $initial(v_{Place,i})$ (for $i > 0$).
The pair (G'_{CCS}, G'_{PN}) is in $\mathcal{R}_{CCS2PN}$ since (1) the syntactic structure is kept (see Proposition 6.5.1), depending on the condition of the path the node v_{Proc} belongs to. If v_{Proc} belings to a path condA, then $v_N = v_{Seq}$. Due to Proposition 6.5.5(A) there exists an initial place v_{Place}, such that $cn1(v_N, v_{Place})$. Since all places marked as initial have a connection to a Token-node then the

pair of the graphs (G'_{CCS}, G'_{PN}) satisfies (2). Similarly, if v_{Proc} belongs to the path condC (due to Proposition 6.5.5(B)) then the condition (4) holds; the condition (3) is unchanged.

$r_1 =$ **ccsSummation.** In this case, we have an invisible step on the CCS-side. If r_1 (see Figure 6.20) is applicable to G_{CCS}, then there are nodes v_{Sum}, v^1_{Proc}, v^2_{Proc} such that $current(v_{Sum}) \wedge subprocess(v_{Sum}, v^1_{Proc}) \wedge subprocess(v_{Sum}, v^2_{Proc})$. Further we consider two paths $\{v^1_i\}^{N_1}_{i=1}$ and $\{v^2_i\}^{N_2}_{i=1}$, such that $\exists j_1, j_2 : v^1_{j_1} = v^2_{j_2} = v_{Sum}$, then obviously $v^1_{Proc} = v^1_{j_1+1}$ and $v^2_{Proc} = v^2_{j_2+1}$. Applying the rule r_1 leads to a graph G'_{CCS} where $current(v^1_{Proc}) \wedge current(v^2_{Proc}) \wedge \neg current(v_{Sum})$.

By (2)-(4) there exists a Place-node v_{Place} in G_{PN} such that $token(v_{Place})$. By Proposition 6.5.4 three cases are possible, depending on which condition the paths $\{v^1_i\}^N_{i=1}$ and $\{v^2_i\}^N_{i=1}$ satisfy.

1. The condition condA holds for both paths, then due to our assumption that $(G_{CCS}, G_{PN}) \in \mathcal{R}_{CCS}$, we have $cn1(v^1_{N_1}, v_{Place})$ and $cn1(v^2_{N_2}, v_{Place})$. Then (2) holds, because $current(v^1_{j_1+1}) \wedge current(v^2_{j_2+1}) \wedge token(v_{Place})$, it means that in both paths condA for every Process-node, which is connected with a Current-node, there exists a Place-node connected with a Token-node (the Current-node was duplicated for each path, therefore the conditions still hold after the rule being applied). The reverse statement (2) for a Place-node holds too. Conditions (3) and (4) are unchangeable in this case. (1) holds since the syntactic structure was not change, when the ccsSummation rule was applied.
2. The condition condC holds for both paths, then due to Proposition 6.5.5 we have $initial(v_{Place})$. Then (4) holds since $current(v^1_{j_1+1}) \wedge current(v^2_{j_2+1}) \wedge token(v_{Place})$ (the Current-node was duplicated for each path, therefore the conditions still hold after the rule being applied). (2) and (3) are unchangeable. (1) holds since the syntactic structure was not change, when the ccsSummation rule was applied.
3. The case, when one of the paths satisfies the condition condA and another – the condition condC, is a mixture of the previous two cases.
4. The condition condB holds for both paths, then $cn2(v^1_1, v_{Place})$ and $cn2(v^2_1, v_{Place})$. (3) holds since the Current-nodes were moved within the paths condB, where on both paths there exists a Process-node which has a corresponding Place-node connected with a Token-node. The reverse statement holds also for the Place-node v_{Place}. (1),(2) and (4) are unchangeable.
5. The case when one of the paths satisfies the condition condA and another the condition condB, is a mixture of the first and second cases .
6. The case when one of the paths satisfies the condition condB and another the condition condC, is a mixture of the second and fourth cases.

The pair (G'_{CCS}, G_{PN}) is thus in $\mathcal{R}_{CCS2PN}$ and furthermore $G_{PN} \xrightarrow{\tau} G'_{PN}$, which completes the proof for this case.

$r_1 =$ **ccsComposition.** In this case, we have an invisible step on the CCS-side. The proof is similar to the case $r_1 =$ ccsSummation.

$r_1 =$ **ccsSequenceA.** Since r_1 (see Figure 6.17) is applicable to a graph G_{CCS}, then there is a match, which consists of the following nodes: v_{Event}, v_{Seq}, v_{Proc}, v_{Curr} and v_{Mark}, and the following conditions hold:

$$subprocess(v_{Seq}, v_{Proc}) \wedge current(v_{Curr}, v_{Seq}) \wedge$$

$$mark(v_{Mark}, v_{Curr}) \wedge event(v_{Seq}, v_{Event}) \wedge name(v_{Event}) = a$$

It means that there exists a path $\{v_i\}_{i=1}^{N}$ such that $v_N = v_{Seq}$ and the condition condA holds. Due to Proposition 6.5.4 (A) there exists v_{Place} in G_{PN} such that $cn1(v_{Seq}, v_{Place})$. Due to our assumption (2) all v_{Place} such that $cn1(v_{Seq}, v_{Place})$ have connection with some Token-node $token(v_{Place})$. Due to Proposition 6.5.2 there exists a corresponding Transition-node v_{Trans} such that $cn(v_{Event}, v_{Trans})$. and $tName(v_{Trans} = a)$. According to the definition of the Petri nets structure, for every Transition-node there exists at least one output Place-node v'_{Place} (see Section 6.2). Therefore, we can build a match for the rule pnMoveTokenA, which moves a Token-node from all input Place-nodes v_{Place} to all output nodes v'_{Place}.

Note that $cn2(v_{Proc}, v'_{Place})$ and in the resulted graphs G'_{CCS} and G'_{PN} we have $current(v_{Proc})$ and $token(v'_{Place})$. Whenever path the Process-node v_{Proc} belongs to, the path satisfies one of the conditions condA, condB or condC, then the conditions (2), (3), (4b) hold for v_{Proc} and v'_{Place} in the graphs G'_{CCS} and G'_{PN}.

However, we must also consider that it could be Process- and Current-nodes $\tilde{v}_{Curr}$ and $\tilde{v}_{Proc}$ in the match for the ccsSequenceA-rule. These nodes satisfy the following conditions:

$$current(\tilde{v}_{Curr}, \tilde{v}_{Proc}) \wedge mark(v_{Mark}, \tilde{v}_{Curr})$$

The ccsSequenceA-rule deletes such nodes. We prove further that this deletion still keepes the resulted graphs G'_{CCS} and G'_{PN} in the relation.

Due to Proposition 6.4.1 for each node $\tilde{v}_{Proc}$ there exists a path $\{\tilde{v}_i\}_{i=1}^{N}$. If there are m number of these nodes, we write that for a node $\tilde{v}_{Proc,m}$ there is a path $\{\tilde{v}_i\}_{i=1,m}^{N_m}$. There exists a common Mark-node v_{Mark} such that $mark(v_{Curr}) \wedge mark(\tilde{v}_{Curr,m})$, that could be caused only by the ccsSummation rule (see Figure 6.17) applied before to the graph G_{CCS}. This means that there are Summation-nodes $\tilde{v}_{Sum,m}$ on every path $\{\tilde{v}_i\}_{i=1,m}^{N_m}$ such that $\exists j_1 < N_1, \ldots, j_m < N_m, \ldots \forall m : (\tilde{v}_{j_m,m} = \tilde{v}_{Sum,m})$.

The node v_{Mark} and adjacent Current-nodes could be removed only by the ccsSequenceA or by ccsCoActionA rule. Since the rules from $\mathcal{RS}_{CCS}$ move the Current-nodes from the root to the leaves, it means that no ccsSequenceA no ccsCoActionA rules were applicable to the nodes $v \in \{\tilde{v}_{j_i,m}, \dots, \tilde{v}_{Proc,m}\}$. Therefore, these nodes v are either Summation-, Composition-, Recursion- or Variable-nodes, but not a Sequence-node.

$$\begin{tikzcd}
 & \tilde{v}_{Sum,1} = \tilde{v}_{j_1,1} \arrow[dl, "\dots"'] \arrow[dr, "\dots"] & & \\
\tilde{v}_{Proc,1} & & \tilde{v}_{Sum,2} = \tilde{v}_{j_2,2} \arrow[dl, "\dots"'] \arrow[dr, "\dots"] & \\
 & \tilde{v}_{Proc,2} & & v_{Seq} = v_N
\end{tikzcd}$$

Due to Proposition 6.5.4 there are three cases for each node $\tilde{v}_{Proc,m}$. We consider one $\tilde{v}_{Proc,m}$, because the proof for the other nodes is the same, and three possible cases for this node.

- $\tilde{v}_{Proc} \in$ condA$(\{\tilde{v}_i\}_{i=1}^{N})$, i.e. we have the case (A) of Proposition 6.5.4, that means that there exist $\tilde{v}_{Place} : cn1(\tilde{v}_N, \tilde{v}_{Place}) \wedge$ $token(\tilde{v}_{Place})$ and $v_{Trans} : cn(\tilde{v}_{Event}, \tilde{v}_{Trans}) \wedge source(\tilde{v}_{Trans}, \tilde{v}_{Place})$. When the Current-nodes were moved through the paths $\{\tilde{v}_i\}_{i=1,m}^{N}$, the Token-node was not moved, because the invisible rules ccsComposition, ccsSummation, ccsVariableDeclaration or ccsRecursion were applied. So after the ccsSequenceA rule deletes the nodes $\tilde{v}_{Curr}$, the pnMoveTokenA deletes Token-nodes from the Place-node $\tilde{v}_{Place}$. Thus, the conditions (1)-(4) hold for the Place-nodes $\tilde{v}_{Place}$ and the Process-node $\tilde{v}_{Proc}$, since the nodes are no longer connected with dynamic elements.

- $\tilde{v}_{Proc} \in$ condB$(\{\tilde{v}_i\}_{i=1}^{N})$, i.e. we have the case (B) of Proposition 6.5.4, that means that there exists a Place-node $\tilde{v}_{Place} : cn2(\tilde{v}_1, v_{Place})$, but $\tilde{v}_1 = v_1$. It means that after the ccsSequenceA rule deletes the node $\tilde{v}_{Curr}$ from $\tilde{v}_{Curr}$, the pnMoveTokenA deletes Token-nodes from the Place-node $\tilde{v}_{Place}$. Thus, the conditions (1)-(4) hold for the Place-nodes $\tilde{v}_{Place}$ and the Process-node $\tilde{v}_{Proc}$, since the nodes are no longer connected with dynamic elements.

- $\tilde{v}_{Proc} \in$ condC$(\{\tilde{v}_i\}_{i=1}^{N})$, is similar to the case condB.

Thus,

$$G_{PN} \xrightarrow{\widehat{\mathsf{pnMoveTokenA}}} G'_{PN}$$

and furthermore $(G'_{CCS}, G'_{PN}) \in \mathcal{R}_{CCS2PN}$.

$r_1 =$ ccsCoActionA. The proof is similar to the case $r_1 =$ ccsSequenceA.

$r_1 =$ ccsRecursion. In this case, we have an invisible step on the CCS-side. If the rule is applicable, then a Current-node is moved from a Variable-node to the Process-node v_{Proc} such that $\exists v_{Rec} : subprocess(v_{Rec}, v_{Proc}) \wedge\ prefix(v_{Proc}, v'_{Var}) \wedge$ $name(v'_{Var}) = pName(v_{Var})$. v_{Var} is a part of the paths that the Process-node v_{Proc} belongs to.

$r_1 =$ ccsVariableDeclaration. In this case, we have an invisible step on the CCS-side. The rule moves a Current-node from a Recursion-node v_{Rec} to the next Process-node v_{Proc}, such that $subprocess(v_{Rec}, v_{Proc})$. According to the TGG rules from MT_{CCS2PN} there is the only one Process-node for each Recursion-node such that $subprocess(v_{Rec}, v_{Proc})$. The ccsVariableDeclaration rule moves a Current-node along the same paths, where the conditions of the relation $\mathcal{R}_{CCS2PN}$ hold for Process-nodes from that path. Since the Petri nets graph G_{PN} is unchangeable, the graph G'_{CCS} is still in the relation with the graph G_{PN}.

Reverse direction: assume $G_{PN} \xrightarrow{r_2} G'_{PN}$. We need to show that there is some G'_{CCS} such that $G_{CCS} \overset{\widehat{r_2}}{\Longrightarrow} G'_{CCS}$ and $(G'_{CCS}, G'_{PN}) \in \mathcal{R}_{CCStoPN}$. Again, we argue on the level of LTSs before renaming.

$r_2 =$ pnInitial. Let $\langle L_1, R_1, \mathcal{N}_1 \rangle$ be the pnInitial rule (see Figure 6.29). If r_1 is applicable in G_{PN}, we have a match $m_1 : L_1 \to G_{PN}$, where there is at least one Place-node $v_{Place,i}$ such that $initial(v_{Place,i})$, where $i \geq 1$. Due to Proposition 6.5.6 there exists a path of Process-nodes $\{v_n\}_{n=1}^{N}$ such that $v_1 = v_{Proc}$ and either the condition condA or condC hold for $\{v_i\}_{i=1}^{N}$. Additionally, for a Process-node v_1 is a root node of a graph G_{CCS}. Therefore we can construct a match in the graph G_{PN} for the ccsInitial rule defined as $\langle L_2, R_2, \mathcal{N}_2 \rangle$ (both being mapped to Initial by map_{CCS} and map_{PN}).
This match $m_2 : L_2 \to G_{PN}$ maps a root Process-node v_{Proc} such that $root(v_{Proc})$.
Thus, ccsInitial is applicable in G_{CCS}. Once the rule is applied, we have a graph G'_{CCS} with one Current-node such that $current(v_{Proc})$ and a graph G'_{PN} with Token-nodes such that $token(v_{Place,i})$ and $initial(v_{Place,i})$.
The pair (G'_{CCS}, G'_{PN}) is in $\mathcal{R}_{CCS2PN}$ since depending on what property in Proposition 6.5.6 holds for v_{Place} ((A) or (B) or both), the nodes either satisfy conditions (2) and (4); (1) the syntactic structure is kept (see Proposition 6.5.1), the condition (3) is unchanged.

$r_2 =$ pnMoveTokenA. Since the pnMoveTokenA rule is applicable in G_{PN}, then there are the following nodes in the graph G_{PN}: v_{Trans} and the nodes $v_{Place,i}$ and $v'_{Place,j}$, where $i, j \geq 1$, in G_{PN} such that for all i, j, $token(v_{Place,i}) \wedge$ $source(v_{Trans}, v_{Place,i}) \wedge\ target(v_{Trans}, v_{Place,j}) \wedge\ tName(v_{Trans}) = a$.
Due to Proposition 6.5.2 there is a unique node v_{Event} in G_{CCS} such that $cn(v_{Event}, v_{Trans})$ and $name(v_{Event}) = a$.

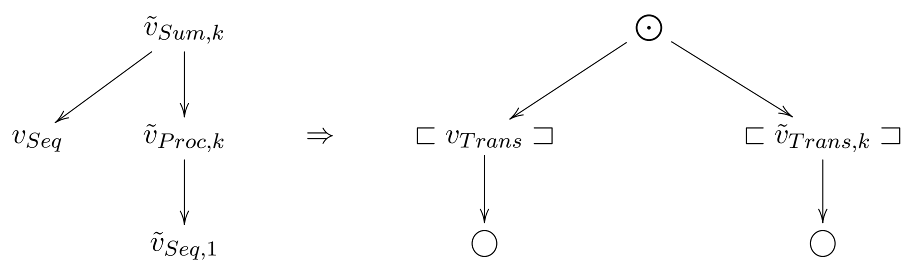
By Proposition 6.4.2 there are nodes v_{Proc} and v'_{Proc} such that $\forall i, j$ we have $cn1(v_{Proc}, v_{Place,i})\wedge$ $cn2(v'_{Proc}, v'_{Place,j})$.
Due to our assumption that $(G_{PN}, G_{CCS}) \in \mathcal{R}_{CCS2PN}$ the condition (2) holds that means that $\exists j : v_j \in \{v_i\}_{i=1}^{N}\wedge$ $current(v_j)\wedge$ $\mathsf{condA}(\{v_i\}_{i=1}^{N})\wedge$ $cn2(v_N, v_{Place,i})$. Then either $j = N$ that means that v_j is a Sequence-node (case (I) below) or $j \neq N$ then by Proposition 6.4.5 v_j is either a Summation-, Composition-, Recursion- or Variable-node (cases (II)-(V)). Hereby, five possible cases are considered below:

(I) $v_j = v_N$ is a Sequence-node. Then due to the facts $current(v_N)$, the tggSequence rule (see Figure 6.17) and Proposition 6.4.6 we can build a match for the ccsSequenceA rule. It could be also possible that in graph G_{CCS} there are additional nodes, such that the application of the ccsCoActionA rule (see Figure 6.19) is possible. We consider further both cases.

- **The** ccsSequenceA **rule** is applicable, then the rule deletes Current-node pointing to the Sequence-node and creates another Current-node for the Process-node v'_{Proc}. Due to Proposition 6.4.1 v'_{Proc} belongs to one or more paths, which satisfy to one of the following conditions: condA, condB, condC. (A) $v'_{Proc} \in \mathsf{condA}(\{v'_i\}_{i=1}^{N})$, then there exists $v_{Seq} = v_N$. Then there is an Event-node v'_{Event} such that $event(v_{Seq}, v_{Event})$. By Proposition 6.5.2 there exists a Transition-node v'_{Trans}. Due to Proposition 6.5.7 there is the only Place-node v'_{Place} such that $source(v_{Trans}, v'_{Place})$ and $target(v'_{Trans}, v'_{Place})$. Then the condition (2) obviously hold, (1) is unchangeable and the satisfaction of conditions (3)-(4) depends on if the node v'_{Proc} belongs to the other paths. (B) $v'_{Proc} \in \mathsf{condB}(\{v'_i\}_{i=1}^{N})$ then $cn2(v'_{Proc}, v'_{Place})$, $current(v'_{Proc})$ and $token(v'_{Place})$ that means that (3) holds, (1) is unchangeable and the satisfaction of conditions (3)-(4) depends on if the node v'_{Proc} belongs to the other paths. (C) $v'_{Proc} \in \mathsf{condC}(\{v'_i\}_{i=1}^{N})$ then $cn2(v'_{Proc}, v'_{Place})$,$current(v'_{Proc})$ and $token(v'_{Place})$ that means that (4b) holds, (1) is unchangeable and the satisfaction of conditions (2)-(3) depends on if the node v'_{Proc} belongs to the other paths.
  However there could be other Current-nodes in the match for the ccsSequenceA rule, which will be deleted. We analyse if the graph G'_{CCS} is still in the relation with the graph G'_{PN}. The Current-nodes from the match for the ccsSequenceA rule are the following:
  $\exists k \geq 0, \tilde{v}_{Curr,k} : mark(v_{Mark}, \tilde{v}_{Curr,k})\wedge$ $current(\tilde{v}_{Curr,k}, \tilde{v}_{Proc,k})\wedge$ $mark(v_{Mark}, v_{Curr})$. Due to Proposition 6.4.1 the Process-nodes $\tilde{v}_{Proc,k}$, where $k \geq 0$, belong to the paths $\{\tilde{v}'_i\}_{i=1,k}^{N}$. The common Mark-node for the nodes v_{Curr} and $\tilde{v}_{Curr,k}$ is possible if there are Summation-nodes $\tilde{v}_{Sum,k}$ on every path $\{\tilde{v}'_i\}_{i=1,k}^{N}$ such that

$\exists j_1, j_2, \ldots \forall k : (\tilde{v}_{j_m,k} = \tilde{v}_{Sum,k})$ (see also case (II)).
The node v_{Mark} and adjacent Current-nodes could be removed from the path only by the ccsSequenceA or ccsCoActionA rules. Since the rules from $\mathcal{RS}_{CCS}$ move the Current-nodes from the root to the leaves, no ccsSequenceA no ccsCoActionA rules were applicable to the nodes $v \in \{\tilde{v}_{j_m,k}, \ldots, \tilde{v}_{Proc,k}\}$. Therefore, these nodes v are either Summation-, Composition-, Recursion- or Variable-nodes, but not a Sequence-node.
Depending on what path the nodes $\tilde{v}_{Proc,k}\}$ belong to, we explain why the conditions (1)-(4) still hold for the resulting graphs G'_{CCS} and G'_{PN}.

a) $\tilde{v}'_{Proc,k} \in \mathsf{condA}(\{\tilde{v}_i\}_{i=1,k}^{N_k})$. It means that there exists $\tilde{v}_{Seq} = \tilde{v}_N$ and $\tilde{v}_{Sum,1} = \tilde{v}_m = v_l$. Since v_i and v_j, where $l < i < N$ and $m < j < N_k$, then the corresponding Petri net structure is as it is shown below.



The Transition-node v_{Trans} fires the token, when the pnMoveTokenA rule is applied. The Current-node is also removed from the node $\tilde{v}_{Proc,k}$, when the ccsSequenceA rule was applied. It means that the Current-node was removed from the path $\tilde{v}'_{Proc,k}$ and the Token-node was deleted from $v_{Place} : cn1(\tilde{v}_{Proc,k}, v_{Place})$. It means that (2) holds and the other conditions are unchangeable.

b) $\tilde{v}'_{Proc,k} \in \mathsf{condB}(\{\tilde{v}_i\}_{i=1,k}^{N})$. It means that there exists $\tilde{v}_{Nil} = \tilde{v}_N$ and $\tilde{v}_{Sum,1} = \tilde{v}_m = v_l$. Since v_i and v_j, where $l < i < N$ and $m < j < N_k$, are not Sequence-node then the corresponding

Petri net structure is as it is shown below.

$$\begin{tikzcd}
& v'_{Seq} \arrow[d] & & & \sqsubset v'_{Trans} \sqsupset \arrow[d] \\
& \tilde{v}_{Sum,k} \arrow[dl] \arrow[dr] & & & \odot \arrow[d] \\
v_{Seq} & & \tilde{v}_{Proc,k} \arrow[d] & \Rightarrow & \sqsubset v_{Trans} \sqsupset \arrow[d] \\
& & \tilde{v}_{Nil,k} & & \bigcirc
\end{tikzcd}$$

The Transition-node v_{Trans} fires the token, when the pnMoveTokenA rule is applied. The Current-node is also removed from the node $\tilde{v}_{Proc,k}$, when the ccsSequenceA rule was applied. It means that the Current-node was removed from the path $\tilde{v}'_{Proc,k}$ and the Token-node was deleted from $v_{Place} : cn2(\tilde{v}_{1,k}, v_{Place})$. It means that (3) holds and the other conditions are unchangable.

c) $\tilde{v}'_{Proc,k} \in \mathsf{condC}(\{\tilde{v}_i\}_{i=1,k}^{N_k})$. It means that there exists either $\tilde{v}_{Var} = \tilde{v}_{N_k}$ and $\tilde{v}_{Rec} = \tilde{v}_j$, where $0 < j < N_k$, or $\tilde{v}_{Nil} = \tilde{v}_{N_k}$. It also means that we have $\tilde{v}_{Sum,1} = \tilde{v}_m = v_l$. Since v_i and v_j, where $l < i < N$ and $m < j < N_k$, are not Sequence-nodes, then the corresponding CCS graph (two cases are possible) and Petri net structures are as it is shown below.

$$\begin{tikzcd}
& \tilde{v}_{Sum,k} \arrow[dl] \arrow[d] & & \tilde{v}_{1,k} \arrow[d] & & \\
v_{Seq} & \tilde{v}_{Rec,k} \arrow[d] & & \tilde{v}_{Sum,k} \arrow[dl] \arrow[d] & & \odot \arrow[d] \\
& \tilde{v}_{Proc,k} \arrow[d] & v_{Seq} & \tilde{v}_{Proc,k} \arrow[d] & \Rightarrow & \sqsubset v_{Trans} \sqsupset \arrow[d] \\
& \tilde{v}_{Var,k} & & \tilde{v}_{Nil} & & \bigcirc
\end{tikzcd}$$

The Transition-node v_{Trans} fires the token, when the pnMoveTokenA rule is applied. The Current-node is also removed from the node $\tilde{v}_{Proc,k}$, when the ccsSequenceA rule was applied. It means that the Current-node was removed from the path $\tilde{v}'_{Proc,k}$ and the Token-node was deleted from

$v_{Place} : cn2(\tilde{v}_{1,k}, v_{Place})$. It means that (3) holds and the other conditions are unchangable.
In summary, we get in the renamed LTS

$$G_{CCS} \xRightarrow{\widehat{\mathsf{ccsSequenceA}}} G'_{CCS}$$

- **The** ccsCoActionA **rule** is applicable. The proof is similar to the previous case.

(II) $j \neq N$ and v_j is a Summation-node. Then due to Proposition 6.4.5(A) and Proposition 6.4.6 it is possible to build a match for the ccsSummation rule, which is mapped to the invisible step. That leads to a graph G''_{CCS} in which there are two Process-nodes v_{j+1} and v'_{j+1} such that $subprocess(v_j, v_{j+1})$, $subprocess(v_j, v'_{j+1})$, $current(v_{j+1})$ and $current(v'_{j+1})$. In addition, there exist the following dynamic nodes v_{Mark}, v_{Curr} and v'_{Curr} such that $mark(v_{Mark}, v_{Curr})$ and $mark(v_{Mark}, v'_{Curr})$.
Due to Proposition 6.4.1 there are paths for each of the Process-nodes v_{j+1} and v'_{j+1}, these paths have the same properties as the paths that the Process-node v_j belongs to. Therefore the conditions (1)-(4) are unchangeable in this case and $(G'_{CCS}, G_{PN}) \in \mathcal{R}_{CCS2PN}$.
If $j+1 \neq N$ then we consider the cases (II)-(V) again, otherwise (I). In summary, we get in the renamed LTS the invisible step:

$$G_{CCS} \xrightarrow{\tau} G'_{CCS}$$

(III) $j \neq N$ and v_j is a Composition-node. Due to Propositions 6.4.5(A) and 6.4.6 we can build a match for the ccsComposition rule, which is invisible. Then the proof procedes similar to the previous case (II).

(IV) $j \neq N$ and v_j is a Recursion-node. Due to Proposition 6.4.5(B) we can build a match for the ccsVariableDeclaration rule, which is invisible. It means that there is a Recursion-node v_{Rec} such that $current(v_{Rec})$. Due to Proposition 6.4.5(B) there is a Process-node v_{Proc} such that $process(v_{Rec}, v_{Proc})$. In a resulted graph G''_{CCS} the Current-node was moved from v_{Rec} to v_{Proc}, i.e. $current(v_{Proc})$ holds.
Due to Proposition 6.4.1 there are paths for the Recursion-node v_{Rec} for which the conditions (1)-(4) hold (because of our assumption that $(G_{CCS}, G_{PN}) \in \mathcal{R}_{CCS2PN}$). The Process-node v_{Proc} belongs to the same paths as v_{Rec}. Since the Petri net graph stays unchangable, the conditions (1)-(4) hold for the graphs G'_{CCS} and G_{PN}. If $j+1 \neq N$ then we consider the cases (II)-(V) again, otherwise (I). In summary, we get in the renamed LTS the invisible step:

$$G_{CCS} \xrightarrow{\tau} G'_{CCS}$$

(V) $j \neq N$ and v_j is a Variable-node. Due to Proposition 6.4.5(C) there exists a Recursion-node v_{Rec} such that $prefix(v_{Rec}, v'_{Var}) \wedge pName(v_{Var}) = name(v'_{Var})$. Let us assume that the nodes v_{Rec} and v_{Var} belong to the same path $\{v_i\}_{i=1}^{N}$, it means that $\exists k : v_k = v_{Rec}$, $v_N = v_{Var}$ and the nodes v_i, where $k < i < N$, are not Sequence-nodes. Then $\{v_i\}_{i=1}^{N}$ satisfies the condition condC, it is not possible, because v_{Var} is from the path that satisfies the condition condA.
There exists also a Sequence-node v_{Seq} such that $subprocess(v_{Seq}, v_1)$ (because otherwise v_1 is a root node, that is not possible). It means when the ccsSequenceA-rule was applied to v_{Seq} the edge $mark(v_{Mark}, v'_{Var})$ was deleted and after application of rules ccsSummation and ccsComposition (application of other rules is not possible) the edge was not restored. It means that we can build a match for the ccsRecursion rule in the graph G_{CCS}.
The ccsRecursion rule moves a Current-node to a Process-node v_{Proc} : $subprocess(v_{Rec}, v_{Proc})$, which belongs to the same paths as the node v_{Var}. Since the graph G_{PN} during the application of the ccsRecursion rule is unchangable, the conditions (1)-(4) still hold for the graphs G'_{CCS} and G_{PN}. In summary, we get in the renamed LTS the invisible step:

$$G_{CCS} \xrightarrow{\tau} G'_{CCS}$$

If $j + 1 \neq N$ then we consider the cases (II)-(V) again, otherwise (I).

r_2 = **pnMoveTokenTauA.** The proof is similar to the previous case.

□

6.6 Properties Interpretation for Petri Nets

In this section we illustrate the behavioural properties preservation. For this we consider a concrete example of a vending machine for selling chocolates (Figure 6.48), which we design with the CCS language. Then we use our transformation to receive a Petri net graph. We use two sample behavioural properties, which we verify against the source model, interpret them for the target language and, finally, verify them against the target language.

6.6.1 System Design

Our vending machine for selling chocolates has a few options. It sells two types of chocolates: a big chocolate, which costs two euro coin and a small one, which costs one euro coin. Note that only these coins can be used. When the payment is done, one of the buttons is pressed: 'big' in case two euro coin was given and 'small' in case one euro coin was given. We define the vending machine, denoted as V, with

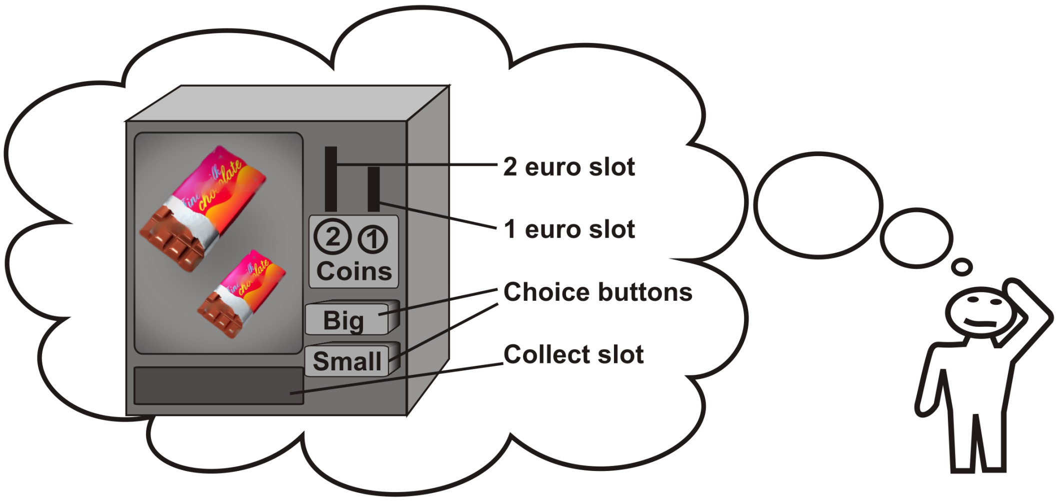


Figure 6.48: Picture of a machine for selling chocolates

the CCS language in terms of its interaction with the environment at its five ports ($twoC$, $oneC$, big, $small$, $colB$, $colS$), as follows:

$$V = \mu x\ (twoC.big.colB.x + oneC.small.colS.x)$$

This means, for example, that to buy a big chocolate you must put in a two euro coin, press the button marked ‘big’, and collect your chocolate from the collect slot. There is also an option to buy a small chocolate, for this you must put in a one euro coin, press the button marked ‘small’, and collect a chocolate from the collect slot. After a chocolate was collected, the process could be repeated from the beginning.

Figure 6.49 depicts a corresponding CCS graph for the process V, denoted G^V_{CCS}. To construct this graph we used patterns for well-formed CCS graphs defined in Subsection 6.1.2. The process V and its subprocesses could be considered as follows:

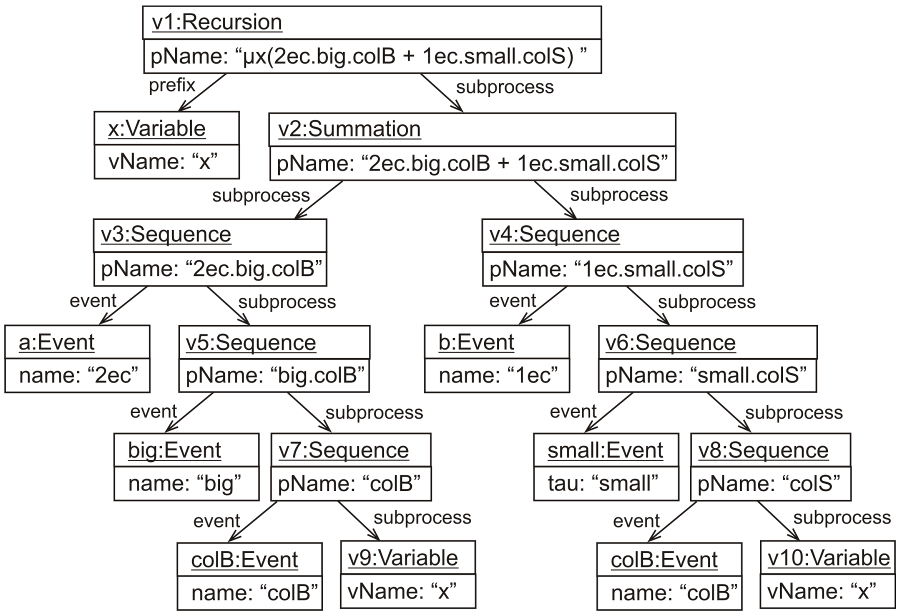


Figure 6.49: The CCS graph for the process V, denoted G^V_{CCS}

$V = \mu x(P)$		- Recursion pattern (E)
$P = P_1 + P_2$		- Summation pattern (B)
$P_1 = 2ec.P_3$	$P_2 = 1ec.P_4$	- Sequence pattern (D)
$P_3 = big.P_5$	$P_4 = small.P_6$	- Sequence pattern (D)
$P_5 = colB.P_7$	$P_6 = colS.P_8$	- Sequence pattern (D)
$P_7 = x$	$P_8 = x$	- Variable pattern (F)

and then each subprocess is turned into a corresponding graph structure.

The Petri net graph in Figure 6.50 is a result of our model transformation $MT_{CCS2PN}(G^V_{CCS}, G^V_{PN})$. Note that we build the CCS and the Petri net graphs simultaneously. We applied step-by-step the following rules: tggInitial, tggVariableDeclaration, tggSummation, tggSequence (6 times) and, finally, tggRecursionI (two times) in order to build a structure of the graph G^V_{CCS}. Then, we projected the result graph on a Petri net meta-model T^{st}_{PN}.

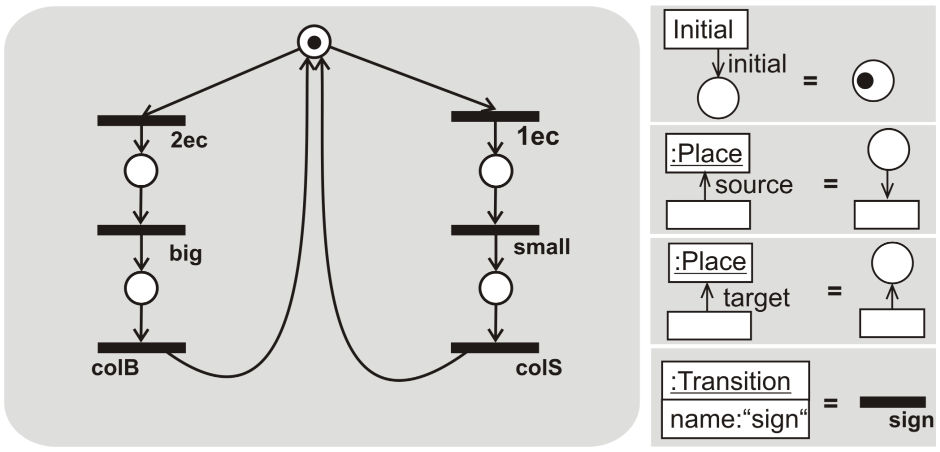


Figure 6.50: The Petri net graph for the process V, denoted G^V_{PN}

6.6.2 Properties Specification

On the next step we want to specify two behavioural properties for the source model. The first one is that the vending machine does not make any loss. Therefore, the first sample property states that the vending machine never gives a big chocolate after it receives one euro coin. We use the names of the graph transformation rules from the CCS semantics $\mathcal{RS}_{CCS}$ in order to specify the required property. Using the ACTL logic we specify the behavioural property with a following formula:

$$\forall G \neg(\mathsf{ccsSequenceOneC}\ U\ \mathsf{ccsSequenceBig}\ U\ \mathsf{ccsSequenceColB}) \tag{6.4}$$

The second behavioural property, we want to verify, states that the vending machine delivers a chocolate each time after it receives a coin. We specify it using the ACTL logic with a following formula:

$$\forall(\mathit{true}\ U\ \mathsf{ccsSequenceTwoC}\ U\ \mathsf{ccsSequenceBig}\ U\ \mathsf{ccsSequenceColB}) \vee$$

$$(\mathit{true}\ U\ \mathsf{ccsSequenceOneC}\ U\ \mathsf{ccsSequenceSmall}\ U\ \mathsf{ccsSequenceColS}) \tag{6.5}$$

An LTS for the CCS graph generated by the semantic system $\mathcal{RS}_{CCS}$ is illustrated in Figure 6.51 in the left.

We verify Formulas (6.4) and (6.5) against the LTS $Q(G^V_{CCS})$. Formula (6.4) holds in the state s_0, as for all paths starting in s_0, there is no run that satisfies the formula ($\mathsf{ccsSequenceOneC}\ U\ \mathsf{ccsSequenceBig}\ U\ \mathsf{ccsSequenceColB}$). Formula (6.4) holds in the state s_0, as there is always a run that satisfies one of the formulas: ($\mathit{true}\ U\ \mathsf{ccsSequenceTwoC}\ U\ \mathsf{ccsSequenceBig}\ U\ \mathsf{ccsSequenceColB}$) or ($\mathit{true}\ U\ \mathsf{ccsSequenceOneC}\ U\ \mathsf{ccsSequenceSmall}\ U\ \mathsf{ccsSequenceColS}$).

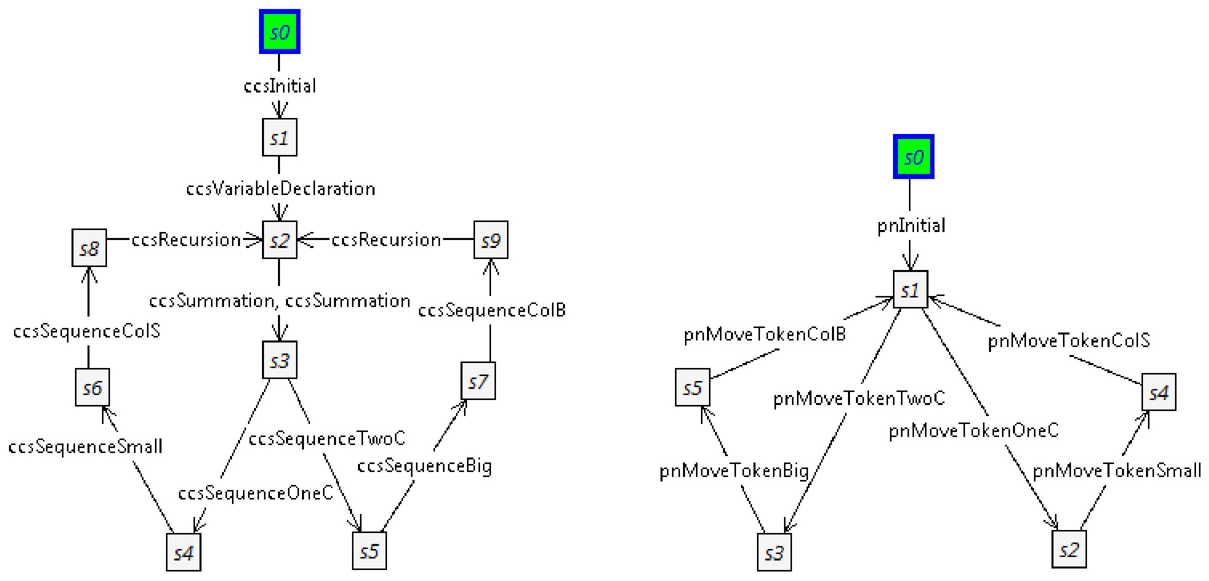


Figure 6.51: LTS $Q(G^V_{CCS})$ for the CCS graph generated by the rule system $\mathcal{RS}_{CCS}$ in the left and LTS $Q(G^V_{PN})$ for the Petri nets graph generated by the rule system $\mathcal{RS}_{PN}$ in the right

6.6.3 Properties Interpretation

In the previous section we showed already that $\forall G_{CCS}$ and $\forall G_{PN}$ such that $MT(G_{CCS}, G_{PN})$

$$Q(G_{CCS}) \approx Q(G_{PN})$$

In Chapter 4 we showed that weak bisimulation implies ACTL equivalence, it means that for any ACTL formula φ follows:

$$Q(G_{CCS}) \models \varphi \quad \Rightarrow \quad Q(G_{PN}) \models \chi(\varphi)$$

Now we want to illustrate, how to interpret (i.e. the meaning of the function χ) the specified behavioural properties for the Petri net language.

We follow the instructions proposed in Chapter 5 Section 5.4. The properties are specified with the graph transformation rules from the rule system $\mathcal{RS}_{CCS}$. The rules from $\mathcal{RS}_{CCS}$ are mapped to the rules from $\mathcal{RS}_{PN}$ (see Section 6.3). According to that mapping we have:

$$\mathsf{ccsSequenceOneC} \mapsto \mathsf{pnMoveTokenOneC}$$

$$\mathsf{ccsSequenceTwoC} \mapsto \mathsf{pnMoveTokenTwoC}$$

$$\mathsf{ccsSequenceBig} \mapsto \mathsf{pnMoveTokenBig}$$

$$\mathsf{ccsSequenceSmall} \mapsto \mathsf{pnMoveTokenSmall}$$

$$\mathsf{ccsSequenceColS} \mapsto \mathsf{pnMoveTokenColS}$$

$$\mathsf{ccsSequenceColB} \mapsto \mathsf{pnMoveTokenColB}$$

We generate an LTS for the graph G^V_{PN} with the rule system $\mathcal{RS}_{PN}$. The result is depicted in Figure 6.51 (in the right).

Finally, we can specify the behavioural properties (introduced in Subsection 6.6.2) over the Petri nets language with the following ACTL formulas:

$$\forall G \neg(\mathsf{pnMoveTokenOneC}\ U\ \mathsf{pnMoveTokenBig}\ U\ \mathsf{pnMoveTokenColB}) \tag{6.6}$$

$$\forall(true\ U\ \mathsf{pnMoveTokenTwoC}\ U\ \mathsf{pnMoveTokenBig}\ U\ \mathsf{pnMoveTokenColB}) \vee$$

$$(true\ U\ \mathsf{pnMoveTokenOneC}\ U\ \mathsf{pnMoveTokenSmall}\ U\ \mathsf{pnMoveTokenColS}) \tag{6.7}$$

We verify Formulas (6.6) and (6.7) against the LTS $Q(G^V_{PN})$. Formula (6.4) holds in the state s_0, as for all paths starting in s_0, there is no run that satisfies the formula ($\mathsf{pnMoveTokenOneC}\ U\ \mathsf{pnMoveTokenBig}\ U\ \mathsf{pnMoveTokenColB}$). Formula (6.4) holds in the state s_0, as there is always a run that satisfies one of the formulas: ($true\ U\ \mathsf{pnMoveTokenTwoC}\ U\ \mathsf{pnMoveTokenBig}\ U\ \mathsf{pnMoveTokenColB}$) or ($true\ U\ \mathsf{pnMoveTokenOneC}\ U\ \mathsf{pnMoveTokenSmall}\ U\ \mathsf{pnMoveTokenColS}$).

6.7 Summary

In the presented case study the idea was to specify a model transformation between the CCS and Petri nets languages. We used our method to specify the languages and the model transformation between them. We used graph transformations for this purpose. Then, we proved the behaviour preservation during the model transformation. At the end we illustrated how the behavioural properties can be specified over the source language and be interpreted for the target language.

CHAPTER 7

Conclusion

In this chapter we summarize this thesis (Section 7.1), the main contribution of which is a method for proving that a model transformation is semantics preserving. The analysis of the method is presented in Section 7.2. A critical look at the achievements and open questions are discussed in Section 7.3. A brief overview of related publications is mentioned in Section 7.4. Finally, Section 7.5 draws future research directions.

7.1 Contribution of this Thesis

The main contribution of this thesis is a five-step method for proving that a model transformation is semantics preserving. In the first two steps, the method gives guidelines on how languages with behavioural semantics must be specified in order to make it possible to define a correct model transformation between them. Correctness in this thesis means preservation of behavioural properties during model transformation, and behavioural semantics of a language is specified by means of operational rules. In the third step, the method provides suggestions on how to define a mapping between the operational rules. In the fourth step, the method describes how to specify a model transformation. In the fifth step, the method explains how to define a relation over the LTSs that is generated for source and target models. Finally, the method shows how to prove that the defined relation is a weak bisimulation.

The second contribution of the thesis is a formal explanation of why weak bisimulation relation implies behavioural properties preservation during the model transformation. For this we performed a proof that the weak bisimulation relation implies ACTL equivalence and illustrated how the ACTL logic allows to specify behavioural properties for a language defined by means of graph transformations. The most interesting detail in this contribution is the interpretation of behavioural properties for the target model.

Finally, the third contribution is a case study, where we defined the model transformation between the CCS language and the Petri nets language and proved that the defined model transformation is semantics preserving. Due to the fact that the method is based on graph transformations, we not only applied our method to the concrete modelling languages between which the behaviour was already studied, but also specified the model transformation with less restrictions on the languages.

7.2 Analysis of the Method

Our method embraces a wide range of different areas, which require a separate discussion. Thus, in this section, we analyse (a) the restrictions on modelling languages in model transformation, (b) the model transformation specification suggested in this thesis, (c) the proof statement for behavioural preserving model transformation, (d) a proposed algorithm within the method to establish weak bisimulation.

7.2.1 Restrictions

There are no strict restrictions on the languages for a model transformation, except that it must be possible to define the syntax of the languages with meta-model and behavioural semantics – by means of graph transformations. Therefore, the method is applicable to a wide range of languages with operational formal semantics.

The method has an additional requirement that is the existence of two non-trivial mapping: between syntactic elements and between semantical rules. This means that the modelling languages must have something in common that is possible to specify, firstly, a syntactical mapping between one or more syntactic elements and, secondly, to identify one or more operational rules from each graph transformation system which perform similar behaviour.

7.2.2 Model Transformation

Additional properties of the model transformation specification proposed in this thesis are discussed in the following.

(Syntactical correctness) The model transformation maps correct models of the source language into correct models of the target language with regards to its specification languages and previously defined syntactical mapping that is guaranteed by the definition of the model transformation over the source and target meta-models.

(Uniqueness) The fact that a model transformation defines a unique target model for a given source model is guaranteed by a syntactical mapping, which maps different elements from source model to different elements of target model. Here, difference means the difference of types.

(Definedness) The fact that is applicable to every model of the source language is true, because the model transformation is specified over the source and target meta-models and engages all elements from both meta-models.

(Understandability) The method is understandable, because it is based on a well-established formalism - graph transformations, - which, firstly, has a visual notation. Secondly, it is one of the popular techniques for capturing model transformations [Var08]. Thirdly, the involved graph notation is very close to the often used UML Class diagram notation that makes graphs easier to understand.

7.2.3 Proof Statement

The method allows to ensure a strong statement, i.e. the preservation of *all* behavioural properties specified with the ACTL for *any* instance of a source model and a target model, which is a result of its model transformation. This statement is strong, because, firstly, the ACTL language is expressive enough to specify many important properties, such as liveness, safety, fairness and so on. It has the same expressive power as CTL*. Secondly, the usage of the method allows to prove the statement once, then the statement about behavioural preservation holds for any transformed instance.

7.2.4 Proof Algorithm

The algorithm for proving that a defined relation is a weak bisimulation is performed in a form of implementation description. Nevertheless, this does not prevent us to estimate its complexity.

(Complexity) The complexity of the performed method is dependent on the sizes of graph transformation systems (which describe the behavioural semantics), since we need to consider $n + m$ cases, where the numbers n and m are the amount of rules in graph transformation systems for the source and target modelling languages. Additionally, the method is strongly dependent on the complexity of the languages, because the correspondences generated by the Triple Graph Grammar (TGG) rules are involved in the proof. These correspondences are based on the syntactic mapping of the source and target languages. The remaining elements are still considered in the proof.

7.3 Discussion of the method

We give an examination of the proposed method by analytical questioning.

▷ *To which extent the problem of semantic gap between modelling languages is solved?*

Semantics of a language has a very broad meaning. In this thesis we considered behavioural semantics, i.e. semantics which prescribes behaviour for each syntactic expression. Then, the behavioural properties are specified over the formal behavioural model. The chosen specification language is powerful enough to specify a wide range of behavioural properties. Thus, the proposed method in this thesis allows to ensure a full preservation of behavioural properties specified over a particular language. However, there are a lot of properties, such as, for example, non-functional properties [LSPS05, GL03], which are not considered in this thesis.

▷ *Is the graph transformation really the best mechanism for the specification of a model transformation?*

The graph transformation technique has a lot of advantages. It is a trustworthy visual formalism, which has gained popularity in recent years. However, there are still some drawbacks. One of them is that the various techniques for graph transformations are not necessarily compatible with each other. Nevertheless, this fact does not affect the applicability of the proposed method, because the main proof is based on comparison of LTSs, which could be generated by all graph transformation techniques. Another drawback is that current tool support is not yet sufficiently mature for industrial use. However, there are working groups (e.g. the Fujaba team [FUJ]), which work in this direction.

▷ *What are the strengths and weaknesses of the method?*

The method is a candidate for solving the problem of big semantic gap in the MDA approach, whose basic idea is to translate an abstract platform-independent model into platform-specific model.

The main weakness of the method that the proof is not automated.

7.4 Overview of Publications

The idea of transforming two real languages - UML Activity Diagrams and TAAL (a Java like language) [EKR$^+$08] - was the starting point for studying behaviour preservation of model transformation.

Proving general correctness of model transformation, i.e. showing that *any* target model exhibits the same behaviour as its source model, is an extremely complex task. This initially hindered us from working on large transformation systems (such as the one between UML Activity Diagrams and TAAL mentioned above), so we started with a small transformation between toy languages [HKR$^+$10a].

7.5 Future Research

There are mainly three directions for future work. The first is connected with encoding of graph transformations into a formal logic. The second direction implies the usage of the method for more complex languages. The third assumes the usage of the case study in the area of refactoring. In the following, we explain each direction in detail.

Our method is based on a proof that a defined relation is a weak bisimulation. This proof is done manually. The ability to control its correctness is not very high. However, the theorem provers, such as Isabelle [NPW02, GH98], propose a solution to specify the given data with high-order logic and, then, to perform a proof automatically. In such cases, the model transformation and meta-models specifications are input data, the observations about the structure of corresponding nodes and the equivalence relation definition are axioms. Then, the process of proving is a standard inference process.

However, the usage of theorem provers involves encoding of graph transformations into a formal logic, which is not obvious. Moreover, the connection between graph transformations and formal logic is not quite studied. There are a lot of open questions. The important one is an encoding of application of graph transformation rule. The proposed methods [Str08, TH09, Pen09] are still far from agreed on to represent a graph as well as transformation rule. Another important question is whether the graph transformation theory completely embedded into the logic theory, still requires a lot of work in order to be answered.

Another direction for future work is an application of the method to a model transformation between relatively complex languages, such as the UML Activity Diagram and the TAAL programming language. The main complication here is that the languages have many elements that are difficult or impossible to map. Therefore, there is a big question as to which extent the proof solves the problem of behavioural preservation during the model transformation between these given languages.

The third direction for future research assumes the usage of the languages from our case study. The CCS language and the Petri net language have a lot in common with some languages used in industry, such as, for example, the Business Process Modelling Language (BPML) and the UML Activity Diagram, respectively. Therefore, the relation between the BPML and Activity diagrams could be studied using our example with respect to a behavioural semantics preservation. The way the model transformation is defined especially allows to study bidirectional model transformation [Ste08].

Bibliography

[AGK09] C. Atkinson, M. Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *The IEEE Transactions on Software Engineering Journal*, 35(6):742–755, 2009.

[AK01] C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on the Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 19–33. Springer-Verlag, 2001.

[AK03] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *The IEEE Software Journal*, 20(5):36–41, 2003.

[AP04] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland, March 2004.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* pub-AW, pub-AW:adr, 1977.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *The Acta Informatica Journal*, 20:207–226, 1983.

[BC87] M. C. Browne and E. M. Clarke. Characterizing Kripke structures in temporal logic. *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 256–270, 1987.

[BCE$^+$99] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. *Handbook of Graph Grammars and Computing By Graph Transformation: Volume III, Foundations.* World Scientific Publishing Co., 1999.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Journal in Theoretical Computer Science*, 59(1-2):115–131, 1988.

[BCM02] P. Baldan, A. Corradini, and U. Montanari. Bisimulation equivalences for graph grammars. In Wilfried Brauer, Hartmut Ehrig, Juhani

Karhumäki, and Arto Salomaa, editors, *Formal and Natural Computing*, volume 2300 of *Lecture Notes in Computer Science*, pages 158–190. Springer-Verlag, 2002.

[BEH06] L. Baresi, K. Ehrig, and R. Heckel. Verification of model transformations: A case study with BPEL. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 183–199. Springer-Verlag, 2006.

[BEMS08] E. Biermann, C. Ermel, T. Modica, and P. Sylopp. Implementing Petri net transformations using graph transformation tools. *Journal in Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, 14, 2008.

[Béz05] J. Bézivin. On the unification power of models. *The Software and System Modeling Journal*, 4(2):171–188, 2005.

[BH02] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. *Journal: Lecture Notes In Computer Science*, 2002.

[BHM09] A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 18 – 33, 2009.

[BK08] C. Baier and J. P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[BKMW09] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What is a multi-modeling language? In *Recent Trends in Algebraic Development Techniques*, pages 71–87, Berlin, Heidelberg, 2009. Springer-Verlag.

[BM10] A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Journal in Formal Aspects of Computing*, 22(3-4):269–296, 2010.

[BNvBK06] D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. *The European Association of Software Science and Technology (ECEASST) Journal*, 1, 2006.

[Boc04] C. Bock. UML 2 activity and action models, Part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.

[BRF$^+$09] P. Barbosa, F. Ramalho, J. Figueiredo, A. Júnior, A. Costa, and L. Gomes. Checking semantics equivalence of MDA transformations in concurrent systems. *Journal of Universal Computer Science*, 15(11):2196–2224, 2009.

[CCN06] B. Y. E. Chang, A. J. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 2006.

[CDSS02] M. Conrad, H. Dörr, I. Stürmer, and A. Schürr. Graph transformations for model-based testing. In Martin Glinz and Günther Müller-Luschnat, editors, *Modellierung*, volume 12 of *LNI*, pages 39–50. GI, 2002.

[CEI$^+$05] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. JVer: A Java verifier. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the Computer Aided Verification, 17th International Conference, CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 144–147. Springer-Verlag, 2005.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CFR08] A. Corradini, L. Foss, and L. Ribeiro. Graph transformation with dependencies for the specification of interactive systems. In Andrea Corradini and Ugo Montanari, editors, *WADT*, volume 5486 of *Lecture Notes in Computer Science*, pages 102–118. Springer-Verlag, 2008.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. *OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM System Journal*, 2006.

[CHM00] A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In *Proceedings of ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques*, 2000.

[Cho57] N. Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[Chr04] A. Christoph. Describing horizontal model transformations with graph rewriting rules. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Proceedings of the conference on Model Driven Architecture: Foundations and Applications (MDAFA)*, volume 3599 of

Lecture Notes in Computer Science, pages 93–107. Springer-Verlag, 2004.

[CMPS82] F. De Cindio, G. De Michelis, L. Pomello, and C. Simone. Milner's communicating systmes and Petri nets. In Anastasia Pagnoni and Grzegorz Rozenberg, editors, *Proceedings of the European Workshop on Applications and Theory of Petri Nets*, volume 66 of *Informatik-Fachberichte*, pages 40–59. Springer-Verlag, 1982.

[CMR$^+$97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - Part I: Basic concepts and double pushout approach. In *Handbook of Graph Grammars* [han97], pages 163–246.

[Cou09] B. Courcelle. Monadic second-order logic for graphs: Algorithmic and language theoretical applications. In Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Proceedings of the 4th International Conference on Language and Automata Theory and Applications (LATA)*, volume 5457 of *Lecture Notes in Computer Science*, pages 19–22. Springer-Verlag, 2009.

[CSP09] V. Chimisliu, C. Schwarzl, and B. Peischl. From UML Statecharts to LOTOS: A semantics preserving model transformation. In Byoungju Choi, editor, *Proceedings of the 9th International Conference on Quality Software (QSIC)*, pages 173–178. IEEE Computer Society, 2009.

[DNM88] P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *The Acta Informatica Journal*, 26:59–91, 1988.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing By Graph Transformation: Volume II, Foundations.* World Scientific Publishing Co., 1999.

[EEL$^+$05] H. Ehrig, K. Ehrig, J. De Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for graph transformation. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 3442:49–63, 2005.

[EEPT06a] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer-Verlag, Berlin Heidelberg, 2006.

[EEPT06b] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Implementation of Typed Attributed Graph Transformation by AGG.* Springer-Verlag, Berlin Heidelberg, 2006.

[EEPT06c] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. *The Fundamenta Informaticae (Fundam. Inf.) Journal*, 74:31–61, 2006.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the Association for Computing Machinery (ACM)*, 33(1):151–178, 1986.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation - Part II: Single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars* [han97], pages 247–312.

[EHK01] G. Engels, R. Heckel, and J. M. Küster. Rule-based specifcation of behavioral consistency based on the uml meta-model. *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 272 – 286, 2001.

[EHRT08] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[EHSW99] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to java. In *Proceedings of the 2nd International Conference on the Unified Modeling Language: beyond the standard*, UML '99, pages 473–488, Berlin, Heidelberg, 1999. Springer-Verlag.

[EK04] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting. *Proceeding of International Conference on Foundations Of Software Science And Computation Structures (FoSSaCS '04)*, 2004.

[EKR$^+$08] G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. From UML Activities to TAAL - towards behaviour-preserving model transformations. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 94–109, Berlin, Heidelberg, 2008. Springer-Verlag.

[EMHL03] G. Engels, J. M.Küster, R. Heckel, and M. Lohmann. Model-based verification and validation of properties. *Electronic Notes in Theoretical Computer Science*, 82:133–150, 2003.

[FKS07] K. Felix, A. Königs, and A. Schürr. Model transformation in the large. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE 2007, pages 285–294, New York, NY, USA, 2007. ACM.

[FUJ] Fujaba Tool Suite 4, University of Paderborn Software Engineering. http://www.fujaba.de.

[GGL$^+$06] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards verified model to code transformations. In *Proceedings of the 3rd Workshop on Model design and Validation (MoDeV2a '06): Perspectives on Integrating MDA and V&V, ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM/IEEE, 2006.

[GGZ$^+$05] L. Grunske, L. Geiger, A. Zündorf, N. V. Eetvelde, P. V. Gorp, and D. Varró. Using graph transformation for practical model driven software engineering. *Journal of Model-driven Software Development*, pages 91–119, 2005.

[GH98] D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. *Journal in Theorem Proving in Higher Order Logics*, number 1479 in Lecture Notes Computer Science:123–142, 1998.

[GK07] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4735 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2007.

[GK10] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *The Software and Systems Modeling (SoSyM) Journal*, 9(1):21–46, 2010.

[GL03] L. Grunske and E. Lück. Application of behavior-preserving transformations to improve non-functional properties of an architecture specification. In Walter Dosch and Roger Y. Lee, editors, *Proceedings of the ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 03), October 16-18, 2003, Lübeck, Germany*, pages 439–445. ACIS, 2003.

[Gle03] S. Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science (UCS)*, 9(3):191–222, 2003.

[GM84] U. Goltz and A. Mycroft. On the relationship of CCS and Petri nets. *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 196–208, 1984.

[GM93] M. Girkar and R. Moll. O(n3) algorithm for bisimulation equivalence w.r.t CTL* without the next-time operator between Kripke structures. Technical report, CiteSeerX - Scientific Literature Digital Library and Search Engine (United States), Amherst, MA, USA, 1993.

[Gol88] U. Goltz. On representing CCS programs by finite Petri nets. In Michal Chytil, Ladislav Janiga, and Václav Koubek, editors, *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS '88)*, volume 324 of *Lecture Notes in Computer Science*, pages 339–350. Springer-Verlag, 1988.

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag New York, Inc., 1979.

[GPR06] V. Gruhn, D. Pieper, and C. Röttgers. *MDA: Effektives Software-Engineering mit UML 2 und Eclipse.* Springer-Verlag, Berlin, 2006.

[GRO] *GRaphs for Object-Oriented VErification (GROOVE) Tool. http://groove.cs.utwente.nl/.*

[GSMD03] P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactoring. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2003.

[han97] *Handbook of Graph Grammars and Computing By Graph Transformation: Volume I, Foundations.* World Scientific Publishing Co., 1997.

[har04] *Meaningful Modeling: What's the Semantics of "Semantics"?*, volume 37. IEEE Computer Society, 2004.

[Hau05] J. H. Hausmann. *Dynamic Meta Modelling.* PhD thesis, University of Paderborn, 2005.

[HHS01] J. H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of UML extensions: An extensible semantics for UML Sequence diagrams. In *Proceedings of the International Symposium on Human-Centric Computing Languages and Environments (HCC)*, pages 80–87. IEEE Computer Society, 2001.

[HKH10] F. Hermann, B. König, and M. Hülsbusch. Specification and verification of model transformations. *Proceedings of the International Colloquium on Graph and Model Transformation*, 2010.

[HKR$^+$10a] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Full semantics preservation in model transformation - a comparison of proof techniques. Technical Report TR-CTIT-10-09, University of Twente, Enschede, February 2010.

[HKR$^+$10b] Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. Full semantics preservation in model transformation - a comparison of proof techniques. In S. Merz D. M'ery, editor, *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM 2010)*, LNCS, pages 183–198, Berlin/Heidelberg, 2010. Springer.

[HKT02] R. Heckel, J. Malte Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery (ACM)*, 32(1):137–161, 1985.

[Hol95] M. Hollenberg. Hennessy-Milner classes and process algebra. In *Proceedings of the Workshop on Modal Logic and Process Algebra*, pages 187–216. CSLI Publications, 1995.

[HV91] J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: a manifesto. *Artificial Intelligence and Mathematical Theory of Computation: papers in honor of John McCarthy*, pages 151–176, 1991.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: a model transformation tool. *Journal: Science of computer programming*, 72(1-2):31–39, June 2008.

[Jen97] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use.* Springer-Verlag New York, Inc., 1997.

[JETE04] J. De L. Jaramillo, C. Ermel, G. Taentzer, and K. Ehrig. Parallel graph transformation for model simulation applied to timed transition Petri nets. *Electronic Notes in Theoretical Computer Science*, 109:17–29, 2004.

[JK09] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer-Verlag New York, Inc., 2009.

[Kö4] J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.

[Kö5] A. König. Model transformation with triple graph grammars. In *Proceedings of the Workshop on Model Transformations in Practice*, 2005.

[Kas06] H. Kastenberg. Towards attributed graphs in Groove: Work in progress. *Electronic Notes in Theoretical Computer Science*, 154(2):47–54, 2006.

[KH08] R. Kastner and T. Huffmire. Threats and challenges in reconfigurable hardware security. In Toomas P. Plaks, editor, *Proceedings of the 2008 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA*, pages 334–345. The CSREA Press, 2008.

[KKR06] H. Kastenberg, A. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 4037:186–201, 2006.

[KN06] G. Karsai and A. Narayanan. Towards verification of model transformations via goal-directed certification. In Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors, *Proceedings of the Second Automotive Software Workshop on Model-Driven Development of Reliable Automotive Services (ASWSD)*, volume 4922 of *Lecture Notes in Computer Science*, pages 67–83. Springer-Verlag, 2006.

[Kot78] V. E. Kotov. An algebra for parallelism based on Petri nets. In Józef Winkowski, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 64 of *Lecture Notes in Computer Science*, pages 39–55. Springer-Verlag, 1978.

[KR06] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. *Proceeding of the 13th International Workshop on Model Checking Software (SPIN 2006)*, 3925:299–305, 2006.

[Kus01] S. Kuske. A formal semantics of uml state machines based on structured graph transformation. In *' '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 241–256, London, UK, 2001. Springer-Verlag.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. Addison-Wesley, 2003.

[Lar88] K. G. Larsen. Proof system for Hennessy-Milner logic with recursion. In Max Dauchet and Maurice Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAP)*, volume

299 of *Lecture Notes in Computer Science*, pages 215–230. Springer-Verlag, 1988.

[LE90] M. Löwe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In Rolf H. Möhring, editor, *Proceedings of the 16rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG '90)*, volume 484 of *Lecture Notes in Computer Science*, pages 338–353. Springer-Verlag, 1990.

[Lei06] J. Leitner. Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. diploma thesis. *Universität Karlsruhe*, 2006.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1&2):181–224, 1993.

[LR10] K. Lano and S. K. Rahimi. Specification and verification of model transformations using uml-rsds. In Dominique Méry and Stephan Merz, editors, *Proceedings of the 8th International Conference on Integrated formal methods*, volume 6396 of *IFM'10*, pages 199–214. Springer-Verlag, 2010.

[LSPS05] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *WORDS*, pages 413–420. IEEE Computer Society, 2005.

[MCBE06] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén. Improvements to combinational equivalence checking. In Soha Hassoun, editor, *Proceedings of the International Conference on Computer-Aided Design (ICCAD'06)*, pages 836–843. ACM, 2006.

[McC04] S. McConnell. *Code Complete, Second Edition.* The Microsoft Press, Redmond, WA, USA, 2004.

[MCG04] T. Mens, K. Czarnecki, and P. V. Gorp. Discussion - a taxonomy of model transformations. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[MDA] Object Management Group: MDA Guide Version 1.0.1. www.omg.org/docs/omg/03-06-01.pdf.

[MEE10] M. Maximova, H. Ehrig, and C. Ermel. Formal relationship between Petri net and graph transformation systems based on functor between M-adhesive categories. *4th International Workshop on Petri Nets and Graph Transformation*, 2010.

[Mil95] R. Milner. *Communication and Concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[MMAB$^+$08] K. D. McDonald-Maier, D. H. Akehurst, B. Bordbar, W. Gareth, and J. Howells. Maths vs. (meta)modelling - are we reinventing the wheel? In José Cordeiro, Boris Shishkov, Alpesh Ranchordas, and Markus Helfert, editors, *Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT)*, pages 313–322. The INSTICC Press, 2008.

[MOF] Meta Object Facility (MOF) 2.0 Core Specification, 2003. http://www.omg.org/spec/mof/2.0/pdf/.

[Mos01] P. D. Mosses. The varieties of programming language semantics. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Proceedings of the Ershov Memorial Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 165–190. Springer-Verlag, 2001.

[MTR05] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes Theoretical Computer Science*, 127(3):113–128, 2005.

[NFGR93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Journal in Computer Networks and ISDN Systems*, 25(7):761–778, 1993.

[NK08] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electronic Notes Theoretical Computer Science*, 211:191–200, 2008.

[NL97] G. C. Necula and P. Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 204. IEEE Computer Society, 1997.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[NR01] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 142–154, 2001.

[NV90] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Proceedings of the LITP Spring School on Theoretical Computer Science, Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[Old86] Ernst-Rüdiger Olderog. Operational petri net semantics for CCSP. In G. Rozenberg, editor, *Proceedings of the European Workshop on Applications and Theory of Petri Nets*, volume 266 of *Lecture Notes in Computer Science*, pages 196–223. Springer-Verlag, 1986.

[OMG] The Object Management Group. http://www.omg.org/.

[Ove97] P. Overell. *Augmented BNF for Syntax Specifications: ABNF.* The RFC Editor, United States, 1997.

[Pad82] P. Padawitz. Graph grammars and operational semantics. *Journal in Theoretical Computer Science*, 19:117–141, 1982.

[PdRV95] A. Ponse, M. de Rijke, and Y. Venema. Modal logic and process algebra: A bisimulation perspective. *Stanford: CSLI Publications*, 1995.

[Pen09] Karl-Heinz Pennemann. *Development of correct graph transformation systems.* PhD thesis, Oldenburg, 2009.

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Journal of Communications of the ACM (JACM)*, 16(8):503–512, 1973.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Technical Report Lecture Notes DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

[PM07] O. Pastor and J. C. Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling.* Springer-Verlag, Berlin, 2007.

[PP91] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. *Journal: SIGPLAN Notices*, 26:274–284, May 1991.

[PR69] J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 609–620, 1969.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Series. Springer-Verlag, 1985.

[Ren04a] A. Rensink. The GROOVE simulator: A tool for state space generation. *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 3062:479–485, 2004.

[Ren04b] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings* *of the International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Berlin, 2004. Springer-Verlag.

[RKE07] G. Rangel, B. König, and H. Ehrig. Bisimulation verification for the DPO approach with borrowed contexts. *Proceeding of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, 2007.

[Roz97] G. Rozenberg. *Handbook of Graph Grammars and Computing By Graph Transformation: Volume I, Foundations.* World Scientific Publishing Co., 1997.

[RPH⁺03] D. Ramljak, J. Puksec, D. Huljenic, M. Koncar, and D. Simic. Building enterprise information system using model driven architecture on J2EE platform. In *Proceedings of the 7th International Conference on TELecommunications (ConTEL)*, volume 2, pages 521–526, 2003.

[RW07] T. Ruhroth and H. Wehrheim. Refactoring object-oriented specifications with data and processes. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 4468 of *Lecture Notes in Computer Science*, pages 236–251. Springer-Verlag, 2007.

[San95] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In Jirí Wiedermann and Petr Hájek, editors, *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer-Verlag, 1995.

[SCDP07] I. Stürmer, M. Conrad, H. Dörr, and P. Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007.

[SCF⁺05] J. S. Sottet, G. Calvary, J. M. Favre, J. Coutaz, A. Demeure, and L. Balme. Towards model driven engineering of plastic user interfaces. In Jean-Michel Bruel, editor, *Proceedings of the Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 191–200. Springer-Verlag, 2005.

[Sch91] A. Schürr. *Operational Specifications with Programmed Graph Rewriting Systems.* PhD thesis, RWTH Aachen, 1991.

[Sch05] H. J. Schneider. Changing labels in the double-pushout approach can be treated categorically. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of

Lecture Notes in Computer Science, pages 134–149. Springer-Verlag, 2005.

[SK08] A. Schürr and F. Klar. 15 years of triple graph grammars. In Ehrig et al. [EHRT08], pages 411–425.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, pages 19–46, 1971.

[Ste08] P. Stevens. Towards an algebraic theory of bidirectional transformations. In Ehrig et al. [EHRT08], pages 1–17.

[Sti95] C. Stirling. Modal and temporal logics for processes. In Faron Moller and Graham M. Birtwistle, editors, *Proceedings of the Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237. Springer-Verlag, 1995.

[Str08] M. Strecker. Modeling and verifying graph transformations in proof assistants. *Electronic Notes Theoretical Computer Science*, 203(1):135–148, 2008.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996.

[TH09] P. Torrini and R. Heckel. Towards an embedding of graph transformation in intuitionistic linear logic. In Filippo Bonchi, Davide Grohmann, Paola Spoletini, and Emilio Tuosto, editors, *Proceedings of the 2nd Interaction and Concurrency Experience: Structured Interactions (ICE)*, volume 12 of *EPTCS*, pages 99–115, 2009.

[UMLa] Object Management Group. Unified Modeling Language, Superstructure v2.0, 2003. http://www.omg.org/cgi-bin/doc?formal/05-07-04.pdf.

[UMLb] Object Management Group. Unified Modeling Language, Superstructure v2.2, 2003. http://www.omg.org/cgi-bin/doc?formal/09-02-03.pdf.

[Val94] R. Valette, editor. *A Term Representation of P/T systems*, volume 815 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994. `http://csd.informatik.uni-oldenburg.de/pub/Papers/cd94-ea.ps.gz` An extended abstract is available on-line.

[Var02] D. Varró. Towards automated formal verification of visual modeling languages by model checking. *SoSyM Journal, Special Section on Graph Tranformation and Visual Modeling Techniques*, 2002.

[Var04] D. Varró. Automated formal verification of visual modeling languages by model checking. *The Software and System Modeling Journal*, 3(2):85–113, 2004.

[Var08] G. Varró. *Advanced Techniques for the Implementation of Model Transformation Systems*. PhD thesis, Budapest University of Technology and Economics, April 2008.

[vKCKB05] M. van Kempen, M. Chaudron, D. Kourie, and A. Boake. Towards proving preservation of behaviour of refactoring of UML models. In *Proceedings of the 2005 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries (SAICSIT '05:)*, pages 252–259, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[VMO] V-Modell Homepage der IABG mbh. http://www.v-modell.iabg.de/.

[Wei09] S. Weißleder. Semantic-preserving test model transformations for interchangeable coverage criteria. In Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung Eingebetteter Systeme V, Schloss Dagstuhl, Germany*, volume 2009-01 of *Informatik-Bericht*, pages 26–35. TU Braunschweig, Institut für Software Systems Engineering, 2009.

[WK05] M. Wimmer and G. Kramler. Bridging grammarware and modelware. *Proceedings of the 4th Workshop in Software Model Engineering (WiSME '05)*, 2005.

List of Figures